SLAC-148
UC-32
(MISC)

# A MODEL FOR DEADLOCK-FREE RESOURCE ALLOCATION*

ROBERT D. RUSSELL

STANFORD LINEAR ACCELERATOR CENTER
STANFORD UNIVERSITY
Stanford, California 94305

PREPARED FOR THE U.S. ATOMIC ENERGY
COMMISSION UNDER CONTRACT NO. AT(04-3)-515

June 1972

Printed in the United States of America. Available from National Technical Service, U. S. Department of Commerce, 5285 Port Royal Road, Springfield, Virginia 22151. Price: Printed Copy $3.00; Microfiche $0.95.

---

* Ph. D. dissertation.

## ABSTRACT

This paper presents an investigation of the phenomenon of deadlock in various types of resource allocation systems. A basic model of resource allocation systems is described, and assumptions about its behavior defined. The problem of deadlock is then identified and discussed in the terms of this formal model, and linear algorithms are developed to detect deadlock in a dynamic manner. A modification of the basic model is then considered to permit a more flexible type of resource control, and the deadlock phenomenon reconsidered in terms of the new system. Again a linear algorithm for deadlock detection is presented. Finally a second modification to the basic model is considered, which permits the linear algorithms already developed to be used in such a way that deadlock is prevented from occurring.

TABLE OF CONTENTS

LIST OF FIGURES

## ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis adviser, Professor William F. Miller, for his guidance, patience and assistance.

I would also like to thank my readers, Professor Donald Knuth, Dr. Thomas Bredt, and Dr. Harry Saal, for their valuable comments and suggestions.

Finally, I would like to thank Mrs. Harriet Canfield for her assistance with the details of seeing this work through to its publication.

## Chapter I. Introduction

Many types of modern computer systems, including large-scale time-sharing systems [25,30,39], multi-programmed remote entry systems [35], and real-time data acquisition and control systems [2,26], can be effectively characterized as a set of asynchronous sequential processes [3,8,21] (tasks, users, jobs) contending for a finite and often very limited set of resources (such as processors, memory, I/O devices, I/O channels, etc.). Although there may be many objectives in designing such systems, one of the more common goals is to maximize the resource utilization as a function of time, since unused resources add to the cost of a system without providing any benefit. A common method of accomplishing this is to process a number of jobs (i.e., resource users) simultaneously, allocating to each only that subset of the total system resource set necessary for the job to make progress. By judicious selection of jobs processed so that their resource requirements complement one another, it is often possible to utilize all (or most) of the system resources simultaneously. This is especially true if the resources required by any one job constitute only a small portion of the resources in the system.

Although quite desirable, multiprocessing systems pose a number of non-trivial problems that arise as "side effects" of the multiprocessing; problems that do not exist in systems that do not attempt concurrent job processing. One of these problems which has come under extensive study recently is that known as "deadlock". Quite simply, a system is deadlocked when the legitimate resource requirements of one or more jobs in the system can never be fulfilled. For example, suppose there are two resource elements $r_a$ and $r_b$ and two jobs $J_1$ and $J_2$. We assume that a job is not required to release control of its

resources until it is finished with them, and that a resource element can be allocated to at most one job at a time. Suppose also that at time t, $r_a$ is allocated to $J_1$ and $r_b$ to $J_2$. If at time $t_1 > t$, $J_1$ requests allocation of $r_b$, it will have to wait until $J_2$ releases control of $r_b$. If at time $t_2 > t_1$, $J_2$ requests allocation of $r_a$, then it will have to wait until $J_1$ releases control of $r_a$. The system is now deadlocked, since $J_1$ is waiting for $J_2$ which is waiting for $J_1$, and this will obviously persist forever.

Dijkstra [8] introduced a detailed discussion of this problem, which he called the "Deadly Embrace" or "Circular Wait" phenomenon. His "Banker's Algorithm" gives a method for preventing deadlock in systems consisting of a set of n independent jobs, $J_1, J_2, \ldots, J_n$, and RMAX independent resource elements that are functionally equivalent. Each job $J_i$ specifies at the time it enters the system a number $M_i$ that is the maximum number of resource elements it will ever need to control at a single time. The actual allocation state of each job at time t is represented by $A_i(t)$, the number of elements currently assigned to $J_i$, and $C_i(t)$, the number of additional elements needed by $J_i$ before it can continue. If $C_i(t)$ is zero, the job is assumed to have all the resources necessary and is expected to proceed at its own rate, independently of and asynchronously to all other jobs in the system. The value $D_i(t) = M_i - A_i(t)$ represents the potential demand of job $J_i$; that is, the number of resource elements it may request at some future time. At any time t the system is free from deadlock if and only if there exists a sequence F of all jobs such that (renumbering the jobs so that $J_k$ is the $k^{th}$ job in F):

$$M_i \leq \text{RFREE}(t) + \sum_{k=1}^{i} A_k(t) \quad \text{for } i = 1, 2, \ldots, n$$

where:

$$RFREE(t) = RMAX - \sum_{k=1}^{n} A_k(t)$$

is the number of resource elements not assigned to any job at time t. The number of operations required by Dijkstra's algorithm to find the sequence F using a repetitive search technique is proportional to $n^2$.

Habermann [11,12] extended Dijkstra's results to the case of m resource classes, each class $R_j$ containing $RMAX_j$ functionally equivalent elements of type j. Each job $J_i$ specifies a maximum demand vector $\overline{M}_i$, such that the $j^{th}$ component of $\overline{M}_i$ is the maximum number of resource elements of type j that might be needed at any single time by $J_i$. Similarly, $A_i(t)$, $C_i(t)$, $D_i(t)$, RFREE(t), and RMAX are extended to m-component vectors in an analogous fashion. The conditions for a system free from deadlock are then the vector equivalents of Dijkstra's conditions, namely the existence of a sequence $F = \{J_1, J_2, \ldots, J_n\}$ such that:

$$\overline{M}_i(t) \leq \overline{RFREE}(t) + \sum_{k=1}^{i} \overline{A}_k(t) \qquad \text{for } i = 1, 2, \ldots, n.$$

Habermann's algorithm for finding this sequence F is also based on a search technique and the number of operations required is proportional to $n^2m$, although he also proved a property of the sequence F that permits the search to terminate early in most cases, thus costing considerably less.

Habermann's results form the basis for subsequent work by Hebalkar [14, 15,16,17], who developed a graph model, and Shoshani [31,32,33], who developed an algebraic model, both giving similar results. These models require

that at entry into the system a job specify considerably more information about its resource requirements than in Habermann's model. Instead of specifying the maximum potential need over the entire lifetime of a job, these models require that a job be broken down into sequential phases such that during each phase the resource requirements remain constant (or decrease only). Although this is a great deal more information for a job to specify in advance, the result is better resource utilization, as is demonstrated quite succinctly by Shoshani. Essentially, the more precise information about resource utilization as a function of time, and the smaller, less demanding job steps enable the scheduler to piece things together better, thereby increasing the resource utilization. However, this increased efficiency is offset by the fact, proved by Hebalkar, that algorithms to perform the allocation cannot be linear in n.

Another method of preventing deadlock was considered by Havender [13], and later Shoshani [31,34]. This method consists of defining a set of restrictions on the legitimate job behavior such that a deadlock becomes impossible. Shoshani has shown that by defining a set of necessary conditions for a deadlock, systems that violate one or more of those conditions are known a priori to be free from deadlock. He gives an example of such a system for parallel access to a data base, and Havender's discussion of OS/MVT provides another example of such a system. This type of "static" deadlock prevention has also been discussed elsewhere by the author [27]. There it is shown that any system which can be represented in terms of a "hierarchical allocation" graph is a priori free from deadlock.

The problem of detecting deadlocks as they occur, rather than trying to prevent them, was investigated somewhat later than Dijkstra's and Habermann's work, although their algorithms for prevention are based on a detection principle.

Shoshani's detection algorithm is essentially a special case of Habermann's algorithm. Murphy [23], however, investigated the problem of deadlock in a system which allows simultaneous sharing of the same resource element by more than one job, although he dealt only with a first come, first serve (FIFO) ordering of requests, and considered the case of only one element per resource class. His detection algorithm for these specialized systems is based on matrix manipulation techniques.

The plan of this thesis is as follows: Chapter II presents a basic model for representing a class of resource allocation systems. The nature of deadlock is defined and discussed more precisely in the terms of this model, and a set of primitives is defined that can be used for scheduling resource allocation. Using this model, linear algorithms capable of detecting deadlock dynamically are presented. Chapter III considers a modification to the model of Chapter II in order to permit simultaneous sharing of resource elements by two or more jobs. This represents a generalization and unification of Murphy's and Habermann's results. A linear deadlock detection algorithm for that system is also presented and discussed. Chapter IV considers a different modification to the model of Chapter II in order to permit deadlock to be prevented rather than just detected. It is shown that by minor adjustments to the basic model, the same linear detection algorithms presented in Chapter II can be used in the new system for deadlock prevention. The last part of Chapter IV applies both previous extensions of the basic model to produce a system with simultaneous resource sharing in which the linear algorithm of Chapter III is used to prevent deadlock dynamically. The models discussed in Chapter IV represent a different departure from Habermann's results than that taken by Shoshani and Hebalkar, who considered systems in which more advance information is

available, whereas we still accept Habermann's minimal assumption of only "worst case" advance information about a job's resource needs, but by careful definition of the scheduler data base and the resource allocation primitives, we are able to make more efficient use of this information to construct practical operating systems with fast algorithms.

This thesis incorporates and extends the work of Habermann, Murphy, and Shoshani with the following aims: (1) defining a general model of a resource allocation system; (2) discussing the problem of deadlock in the formal terms of the model; (3) presenting linear algorithms for dynamic deadlock detection so that it becomes feasible, from the standpoint of computational overhead, to incorporate this framework into operational computer systems; (4) demonstrating how the technique of deadlock detection can be coupled with advance information on resource requirements of jobs to enable deadlock to be prevented; (5) considering systems that permit simultaneous sharing of resource elements between jobs, thereby making the model more attractive as a basis for operational multiprocessing systems.

## Chapter II. Dynamic Deadlock Detection

### A. The Basic Model for System $\hat{S}$

A System $\hat{S}$ consists of a set of m resource classes $\hat{R} = \{R_1, R_2, \ldots, R_m\}$, a set of n jobs $\hat{J} = \{J_1, J_2, \ldots, J_n\}$, a scheduler data base B, and a scheduler S. Each resource class $R_j$ consists of $RMAX_j$ independent and functionally equivalent resource elements of type j ($R_j = \{r_{j1}, r_{j2}, \ldots, r_{jRMAX}\}$). Resource elements are the entities, either hardware, software, or firmware, that perform the basic work functions in the system. They are atomic building blocks of system $\hat{S}$ whose substructure is of no interest in this model. As such, they cannot be created, destroyed, or made to change class membership.

Jobs are the entities in system $\hat{S}$ to which resources are attached for the purpose of performing a desired computation. A job directs and coordinates the actions of the otherwise independent resource elements. Each job is completely independent of all other jobs in the system, and has no way to detect the existence of any other job. Jobs are the resource users, and may be both created and destroyed in the terms of this model.

Both jobs and resource elements are defined intuitively as "sequential processes" [7,8,21], and are describable in terms of a process definition, a process state-vector, and a rule by which the state of the process may change over time. The details of this description are not considered in this model.

The scheduler data base B is a set of data structures representing the status of all jobs and resource elements, including information on the resource requirements and resource allocations of each job. In this model, the scheduler data base B can be considered as the "state-vector" of the entire system $\hat{S}$. Changes in the state of the system are accomplished only by the actions of a hierarchical set of "primitives" acting upon the data structures in B. This set

of primitives defines the scheduler S, and is designed to fulfill four basic requirements necessary for the operation of system $\hat{S}$: (1) as a means by which a job makes known its desire to acquire or release control of resource elements; (2) as a means by which the desires of a job are satisfied in a manner consistent with the operating constraints imposed by the model; (3) as a means for communication between a job and its assigned resources; and (4) as a means by which jobs are created and destroyed. Only primitives necessary for the first two of these functions will be discussed in this thesis.

B. System Operation

Since this model is concerned primarily with the definition and implementation of the scheduler primitives mentioned above, the behavior of individual jobs or resource elements can largely be ignored insofar as any computation or work performed by them is concerned. We are interested only in a very select type of state transition which may be made by the sequential process that defines a job or resource element, and this is the type of transition involving a primitive of scheduler S.

We therefore define two possible states for each job in $\hat{J}$, called active and waiting, and two possible states for each resource element in $\hat{R}$, called free and owned. At any instant of time a process is in exactly one of the two possible states allowed to it, and transitions between the states are caused only by the actions of scheduler primitives. The state diagram for jobs is given in figure II. B. 1, and for resource elements in figure II. B. 2. The edges are labelled with the names of the scheduler primitives that cause the state transition represented by the edge.

A job in the active state has all the resources necessary for it to perform its computation, and it is therefore able to make progress with that computation (by changing its process state-vector in an asynchronous manner not considered by this model) without any assistance from the scheduler. During the course of its activity, however, an active job may reach a state in which further progress is impossible without more resources. At this point the job invokes the REQUEST primitive operation, supplying as a parameter to this primitive a vector $\overline{N} = (N_1, N_2, \ldots, N_m)$, defined such that the $j^{th}$ component, $N_j$, is the number of resource elements of type j being requested. This primitive causes the job to enter the waiting state, where it will remain until the scheduler

decides to satisfy this request. This decision is made by the ASSIGN primitive, and will depend on the particular scheduling algorithm embodied in the primitive definition. This can involve many considerations, but the primary concern of this thesis will be algorithms formulated to detect or prevent deadlock situations arising from satisfaction of a request. For any choice of an ASSIGN primitive, a minimum requirement for choosing a request to be satisfied is that enough resource elements of the requested types be in the free state. (This requirement can be relaxed if simultaneous resource sharing between several jobs is allowed, see Chapter III.) When a request is to be satisfied, the scheduler must invoke the ALLOCATE primitive for each resource element requested in order to select a free element and allocate control of that element to the job, thereby changing the state of the element from free to owned. After all requested allocations are made to the job, the ASSIGN primitive will change the job's state from waiting back to active, thereby enabling the job to proceed once again with its computation.

Whenever an active job determines that it no longer needs control of some of its owned resource elements, it invokes the RELEASE primitive, specifying as a parameter the set of names of resource elements that are no longer needed. This primitive causes the job to enter the waiting state until the resource elements have been deallocated. This is accomplished by the UNASSIGN primitive, which must invoke the DEALLOCATE primitive for each resource element specified by the job in order to remove control of the element from the job, thereby changing the state of the deallocated resource element from owned to free. After all resource elements have been deallocated, ASSIGN will change the state of the job from waiting back to active.

Although not considered in this thesis, it is obvious that some mechanism is necessary to synchronize the control of resource elements and to guarantee the access security of an allocation. This is considered elsewhere by the author and others [7,8,19,22,28,29].

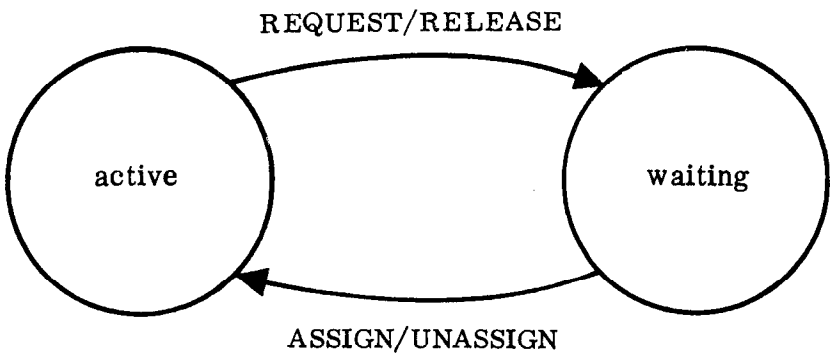


Figure II. B. 1 State Diagram for Jobs in $\hat{J}$.

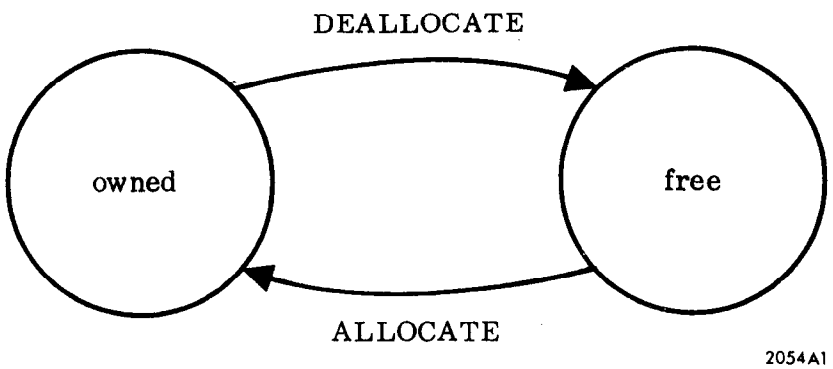


Figure II. B. 2 State Diagram for Resource Elements in $\hat{R}$.

## C. Hierarchy of Primitive Pairs

The scheduler primitives are arranged in a hierarchical set of primitive pairs, such that primitives at one level can invoke other primitives only at a lower level, and, except at the highest level, only one member of a pair is permitted to be "executing" at any instant. The highest level contains the primitive pair REQUEST and RELEASE, and in an extended version of this model, would also contain primitives to initiate and terminate jobs, and primitives to communicate between a job and its controlled resource elements. These are called "job primitives", since they can be invoked only by jobs, and are the only primitives available to a job. In keeping with the notion of a sequential process, a job can be executing only one of these primitives at any instant of time, and the job primitives are designed to be always executable, since all blocking of the progress of a job is done as part of the primitive itself.

The second level in the hierarchy consists of the ASSIGN and UNASSIGN primitive pair. These primitives define the true "scheduling" aspects of the scheduler S, in the sense that these primitives embody the decision rules for determining whether or not REQUESTs and RELEASEs can be satisfied, as well as the algorithms for enforcing any constraints imposed on the scheduler, such as the one that it operate in a deadlock-free manner.

The lowest level contains the ALLOCATE and DEALLOCATE primitive pair, which have already been mentioned as causing the state changes of resource elements and as performing the details necessary for establishing and deleting access control between a job and resource elements. In addition, this level also contains a BLOCK and UNBLOCK primitive pair to perform the details necessary for job state transitions, and an ENQUEUE and DEQUEUE primitive pair, to perform the details necessary for proper queue management.

In an extended model, this level would also contain a primitive pair for attending to the details of job creation and job deletion. All primitives at the lowest level perform basic bookkeeping functions that are necessary for proper functioning of a system. However, the actual details of how these functions are implemented are of no interest in this model (see however [28]), so that primitives at the lowest level will not be further defined. The detailed definitions of the other primitives will be given in the following sections. Figure II. C. 1 summarizes the primitive pair hierarchy and the functions performed by each primitive.

Lowest Level

| | |
|---|---|
| ENQUEUE(J,q) | add job J to queue q |
| DEQUEUE(J,q) | remove job J from queue q |
| BLOCK(J) | change state of job J from active to waiting |
| UNBLOCK(J) | change state of job J from waiting to active |
| ALLOCATE(J,j) | establish access between job J and a free resource element of type j, changing the state of the selected element from free to owned |
| DEALLOCATE(J,r) | remove access of resource element r from job J, and change the state of r from owned to free |

Intermediate Level

| | |
|---|---|
| ASSIGN(i, $\overline{N}$) | attempt to satisfy the demand $\overline{N}$ of job $J_i$ to gain control of the number and type of resource elements represented by the vector $\overline{N}$ |
| UNASSIGN(i,N) | attempt to satisfy the demand of job $J_i$ to release control of the resource elements in set N |

Highest Level (Job primitives, issued by job J

| | |
|---|---|
| REQUEST($\overline{N}$) | prevent job J from proceeding until it obtains access control of the type and number of resource elements specified by the vector $\overline{N}$ |
| RELEASE(N) | prevent job J from proceeding until access control between J and the resource elements named in set N has been removed |

Figure II.C.1 Summary of Scheduler Primitives

D. Intuitive Considerations of Deadlock

Although defined precisely in section II. G, an intuitive definition of deadlock in system $\hat{S}$ at time t is the existence of at least one job that is unable to complete its computation in a finite time due to an inability to obtain needed resources. Therefore, a system will be deadlock-free at time t if (and only if) there exists a sequence of resource allocations, consistent with any constraints on the operation of the system, that will supply all the needed resource elements to all jobs in a finite time. Such a sequence is called a finishing sequence (or an allocation schedule), and is also defined formally in section II. G.

A deadlock detection scheme is a method of ascertaining at any time whether or not a finishing sequence exists for system $\hat{S}$ in its current state. A deadlock prevention scheme is further classified into two categories, called static and dynamic. A static prevention scheme consists of a set of permanent restrictions on the behavior of all jobs in the system so that any permissible sequence of resource allocations from any state of the system can be proved a priori to form a finishing sequence. A dynamic prevention scheme consists of a rule for changing the state of a system that, given any state that is known to have a finishing sequence, will allow only transitions to a next state that also has a finishing sequence. It is assumed that without this rule, it cannot be shown a priori that all possible allocation sequences will also be finishing sequences.

Shoshani [31] has defined a set of conditions that are necessary for the occurrence of deadlock. A static prevention scheme can, therefore, be seen as restrictions to job behavior that cause one or more of these necessary conditions to be absent from the operation of the system. Static deadlock prevention is discussed elsewhere by the author [27] and will not be covered

any further here. The remainder of this chapter and the next one will be devoted to considerations of dynamic deadlock detection. Dynamic deadlock prevention is discussed in Chapter IV as an extension of deadlock detection.

E. Properties of the Model

Inherent in the discussion so far, and in all that is to follow, are five very basic assumptions about the systems being represented in the model.

Assumption 1: (Job Independence) Each job is a unique "sequential process" or "computation" that is independent of, and asynchronous to, all other jobs in the system $\hat{S}$. Each job is totally unaware of the existence of other jobs, and has no way to detect their presence.

Assumption 2: (Non-virtual Resources) A job can request and be allocated at most a maximum of all the resource elements that exist in each resource class. There are no "virtual" resources accounted for in this model.

Assumption 3: (Resource Element Equivalence) Resource elements are grouped into fixed equivalence classes such that each element of a class performs identically as far as a job controlling it is concerned. A job can make requests only for "any element" of a specified type, and must accept the elements chosen by the scheduler primitives. Further, elements of a resource class are uniquely distinguishable, but are unordered, so that there is no notion of "neighboring" or "consecutive" elements within a class (as for example, consecutive pages of core storage).

Assumption 4: (Finite Control) A job can make only a finite number of REQUESTs and RELEASEs. Further, if provided with all requested resources, a job must attain in finite time a state in which all resources controlled previous to that

time are released. Termination of a job implies RELEASE of all resources controlled by the job at that time.

Assumption 5: (Non-shared Resource Elements) A resource element can be controlled by at most one job at any instant of time. There is no simultaneous sharing of a single resource element by two or more jobs.

This last assumption is not necessary in its entirety, since systems in which simultaneous control of a single resource element is allowed can still encounter deadlock problems, as is discussed in the next chapter. However, it simplifies the following discussion to make this assumption now, and then consider the consequences when it is later relaxed. Note however, that if all resource elements were completely shareable at all times, no deadlock could arise. This is obvious, since any request for resources could be immediately satisfied at any time. We state next the modified version of assumption 5, but postpone discussion of it until the next chapter.

Assumption 5': It must be possible for a job to request and be granted control of resource elements in either of two possible modes: exclusive, in which only this job has control of the element; or shared, in which other jobs may also have control of the element at the same time.

With these assumptions in mind, we can delimit some of their implications for the model.

Consequence 1: The only source of delay to the progress of a job is an unsatisfied REQUEST or RELEASE. This follows from the independence of a job from all others, and the fact that the only communication between a job and its environment is by the use of these two scheduler primitives. Therefore, scheduler

actions are the only thing which can prevent a job from making progress. It will be shown later that a RELEASE can always be satisfied immediately upon invocation. Hence only an unsatisfied REQUEST will cause a delay to the real-time progress of a job. This implies that deadlock can be caused only by "poor scheduling", not by the direct interaction of the jobs themselves. It further implies that once the resource scheduler has been guaranteed correct, deadlock in the entire system is eliminated, since there exists no other source of interference to the progress of a job (such as waiting for another job to do something).

Consequence 2: Since they are independent, jobs can be run to completion in an arbitrary order. In particular, a strategy could be chosen such that only one job at a time would be allowed to control any resources. This would prove extremely wasteful of resources, since many would be unused much of the time, but due to assumption 4 and consequence 1, it is guaranteed that jobs can be processed in a strictly sequential manner, and due to assumption 4, resources allocated to that job will always be freed for the next job in the sequence. (This is usually called a "batch processing" type of scheduling.) The occurrence of deadlock is therefore a direct consequence of allowing two or more jobs to compete simultaneously for a limited number of resources.

Consequence 3: In keeping with the assumed independence of jobs, the REQUEST and RELEASE primitives are defined such that they can affect only the status of resource elements allocated to the job invoking the primitive. A job cannot explicitly request or release resources on behalf of another job, nor can it do so implicitly by causing the scheduler to change the allocation of resource elements to other jobs in response to a REQUEST or RELEASE. This eliminates resource preemption, and implies that there is no hierarchy between jobs, in

which one job exercises control over the resource requirements of another subordinated job.

Consequence 4: By construction, only active jobs can invoke primitives, and as stated above, these affect only resource elements allocated to that job. Therefore, once a REQUEST or RELEASE is invoked by an active job, no further REQUESTs and/or RELEASEs affecting that job can occur until the pending REQUEST or RELEASE is satisfied by the scheduler and the job becomes active again. In other words, only the scheduler can reactivate a waiting job, and this can happen only by fulfilling the REQUEST or RELEASE that caused the job to enter the waiting state.

Consequence 5: While in the waiting state a job can retain control of all resource elements previously allocated to it and not yet released.

Consequence 6: There is no necessary ordering by which requests must be chosen for satisfaction, provided that all resource elements necessary to satisfy a single REQUEST are allocated as part of a single ASSIGN. Therefore the choice of a selection rule is left completely open as far as the basic model is concerned.

Consequence 7: There is no restriction on the order in which jobs make requests for resources, and except for the limitations imposed by assumption 2, each REQUEST can be for any number of elements from any resource classes.

These assumptions and consequences are important to the problem of deadlock for several reasons. We must postulate that a job will release control of all its resource elements within a finite time after a finite number of requests have been satisfied since, in general, it is impossible to prove whether or not a computation will indeed attain such a state [36], and moreover, the details of how and when a job will reach such a state are of no interest to this model.

Job termination will not require any special mention, except where specifically noted, since it implies RELEASE for all controlled resource elements of the terminating job, and can therefore be treated under the general considerations of a RELEASE in most of what follows.

If virtual resources could be created whenever needed, no deadlock would be possible since a REQUEST could always be satisfied by simply creating the requested resources. (Other problems may arise, however, such as the thrashing phenomenon discussed by Denning [5,6].) Therefore virtual resources are eliminated from consideration in this model by assumption 2. For the same reason, resource elements which are not available for exclusive control by a single job at a time are eliminated from this model by assumption 5. Consequence 2 implies that deadlock is not inherent in the use of resources by independent jobs, since a trivial allocation scheme always exists, but that it arises from the attempt to increase resource utilization by allocating resource elements to two or more simultaneously progressing jobs. Consequence 4 indicates that, due to the limited type of communication possible between a job and the scheduler, only the actions of the scheduler can cause a deadlock to arise.

Resource preemption would alleviate the problem of deadlock to a large extent, since requests by priority jobs could always be satisfied by "borrowing" resources from lower priority jobs. The problem then reduces to that of detecting or preventing deadlock in the allocation of resources among jobs of equal priority (where preemption is impossible), or to reducing the cost of preemption to a minimum (see [31]). Consequence 3 rules out preemption in this model, either explicit or implicit, thereby requiring the scheduler to "forsee" potential deadlock situations and to avoid them if deadlock is to be prevented, since preemption cannot be used as an "escape".

Consequence 6 permits the system designer to specify a method of selecting REQUESTs to be satisfied in a manner that is consistent with the operating constraints he wishes to impose upon the scheduler. In particular, it will be seen that this ability is essential to guarantee that deadlock does not exist, since an imposed ordering, such as first come, first serve, may create deadlocks that could otherwise be avoided [20,23].

Consequence 7 implies that the jobs operate completely dynamically in a "laissez-faire" environment. The resource needs of a job depend only on that job itself and are decided upon within a job's internal processing, the only restriction from the system environment being that the system resource capacity cannot be exceeded (assumption 2).

## F. Scheduler Data Base $B_1$

This section describes scheduler data base $B_1$, the first version of data base B in the model that will be investigated. A schematic diagram of the data structures comprising $B_1$ is given in figure II. F. 1. Each data item is defined separately, and its function described, in the following.

The following notation will be used throughout. Subscripts are always used to specify a single element of a matrix or vector. The $i^{th}$ row of a matrix A will be indicated by $\overline{A}_i$. Most items in data base B can be changed by the action of scheduler primitives, and hence are functions of time. All functional dependence will be indicated by the use of parentheses ( ), but whenever the meaning is clear, the functional dependence on time, (t), will not be written explicitly.

Item 1: $\overline{RMAX}$--an m-component vector, called the "total resources" vector. The $j^{th}$ component is the number of resource elements that exist in resource class $R_j$. By assumption this number is constant over all time, since resource elements cannot be created, destroyed, or made to change class membership.

To simplify the discussion it is assumed that each resource class $R_j$ contains at least one element, since otherwise any REQUEST for an element in that class could never be satisfied. This implies the following relationship for $\overline{RMAX}$:

$$0 < RMAX_j \quad \text{for all } j = 1, 2, \ldots, m. \tag{II. F. 1}$$

Item 2: $\overline{RFREE}$--an m-component vector, called the "free resources" vector. At any instant of time t, the $j^{th}$ component of $\overline{RFREE}$ is the number of resource elements in class $R_j$ that are in the free state.

Obviously for any instant of time t we must have:

$$0 \leq RFREE_j(t) \leq RMAX_j \quad \text{for all } j = 1, 2, \ldots, m. \tag{II. F. 2}$$

Item 3: A--an n by m matrix, called the "current assignment" matrix. The $i^{th}$ row of A corresponds to job $J_i$ ($1 \leq i \leq n$), and is defined such that at any instant of time t, the $j^{th}$ element of row i is the number of resource elements of type j controlled by job $J_i$ at that time t.

Due to assumption 2 we have the following relationship at all times:

$$0 \leq A_{ij}(t) \leq RMAX_j \qquad \text{for all } i = 1, 2, \ldots, n \text{ and all } j = 1, 2, \ldots, m. \qquad \text{(II. F. 3)}$$

The sum of all elements in the $j^{th}$ column of A is just the total number of resource elements of type j that are controlled by the n jobs in $\hat{J}$. By the definition of $\overline{RMAX}$ we must therefore have at all times:

$$0 \leq \sum_{i=1}^{n} A_{ij}(t) \leq RMAX_j \qquad \text{for all } j = 1, 2, \ldots, m. \qquad \text{(II. F. 4)}$$

But resource elements that are controlled by a job are by definition in the owned state, and since a resource element always must be in exactly one of the two possible states owned or free, at any instant of time t we must also have:

$$RFREE_j(t) = RMAX_j - \sum_{i=1}^{n} A_{ij}(t) \qquad \text{for all } j = 1, 2, \ldots, m. \qquad \text{(II. F. 5)}$$

Item 4: D--an n by m matrix, called the "unsatisfied demand" matrix. The $i^{th}$ row of D corresponds to job $J_i$ ($1 \leq i \leq n$), and is defined such that the $j^{th}$ element in row i is the number of resource elements of type j that have been previously requested by job $J_i$ but have not yet been allocated to that job.

We observe that job $J_i$ is in the active state if and only if all elements in the $i^{th}$ row of D are zero. Since the $i^{th}$ row of A represents the resource

elements already controlled by $J_i$, and the $i^{th}$ row of D represents any additional elements required by $J_i$, assumption 2 can be restated as the following relation, valid for any time t:

$$0 \leq A_{ij}(t) + D_{ij}(t) \leq RMAX_j \qquad \text{for all } j = 1, 2, \ldots, m \text{ and all } i = 1, 2, \ldots, n. \qquad \text{(II. F. 6)}$$

It is important to note that these first four data structures, plus items 9 and 10 discussed later, are the only ones that would be necessary to define in B for the functioning of a set of scheduler primitives S that do not utilize the fast deadlock detection algorithms to be presented later. (In fact, having both $\overline{RMAX}$ and $\overline{RFREE}$ provides an unnecessary redundancy, since one can always be computed from the other by means of relation (II. F. 5).) The following items are data structures that are present in B for the sole purpose of keeping track of information that will be needed by the fast deadlock detection algorithms. As will be shown later, keeping this information at all times up-to-date in the data base B, rather than computing it each time it is needed to check for deadlock, is the key to formulation of the fast algorithms.

Item 5: $\overline{DNUMB}$--an n-component vector, called the "measure of D". At any instant of time t, the $i^{th}$ component of $\overline{DNUMB}$ is the number of positive elements in the $i^{th}$ row of matrix D.

This can be expressed more precisely by first defining the function:

$$\Delta(i,j,t) = \begin{cases} 0 \text{ if } D_{ij}(t) = 0 \\ 1 \text{ if } D_{ij}(t) > 0 \end{cases}$$

then $\overline{DNUMB}$ is defined as:

$$DNUMB_i(t) = \sum_{j=1}^{m} \Delta(i,j,t) \qquad \text{for all } i = 1, 2, \ldots, n.$$

Obviously for any instant of time t the following relationship holds:

$$0 \leq DNUMB_i(t) \leq m \qquad \text{for all } i = 1, 2, \ldots, n. \qquad \text{(II. F. 7)}$$

Item 6: $\overline{RAVAIL}$--an m-component called the "available resources" vector. The $j^{th}$ component of $\overline{RAVAIL}$ is the total number of resource elements of type j that are either in the free state, or are controlled by a job $J_i$ with $DNUMB_i = 0$.

Intuitively, resource elements counted in $\overline{RAVAIL}$ are the only resource elements that are potentially available in a next assignment, since these are the only resource elements that are free or might be released from the control of a job before a new assignment is attempted. (Consequence 4 indicates that resource elements cannot be taken away from jobs that are waiting for an unsatisfied request, i.e., from jobs $J_i$ with $DNUMB_i > 0$.)

The definition of $\overline{RAVAIL}$ can be expressed more precisely by first defining the function:

$$\lambda(i,t) = \begin{cases} 0 \text{ if } DNUMB_i(t) > 0 \\ 1 \text{ if } DNUMB_i(t) = 0 \end{cases}$$

then $\overline{RAVAIL}$ is defined as:

$$RAVAIL_j(t) = RFREE_j(t) + \sum_{i=1}^{n} \lambda(i,t) A_{ij}(t) \qquad \text{for all } j = 1, 2, \ldots, m.$$

Item 7: Q--an n by m matrix, called the "request queue" matrix. The $j^{th}$ column of Q represents a linear queue, or ordered list, of indices of jobs with unsatisfied demands for resource elements of type j. That is, at time t, index i is in column j of Q if and only if $D_{ij}(t) > 0$.

Item 8: $\overline{QSIZE}$--an m-element vector, called the "queue size" vector. The $j^{th}$ element of $\overline{QSIZE}$ is the number of job indices in the $j^{th}$ column of matrix Q.

As discussed in consequence 4, a job can make only one REQUEST or RELEASE at a time, which must be satisfied before a new REQUEST or RELEASE is made. This implies that the index i for job $J_i$ will appear at most once in column j of Q, and then only if $D_{ij}(t) > 0$. Since there are exactly n jobs, we have:

$$0 \leq QSIZE_j(t) \leq n \qquad \text{for all } j = 1, 2, \ldots, m. \qquad \text{(II. F. 8)}$$

Column j of matrix Q may contain anywhere from 0 to n job indices, since there may be unsatisfied demands for resource elements of type j by anywhere from 0 to n jobs, and it is important to note that indices appearing in one column may or may not appear in other columns (depending of course on the job 's demand). In the Q matrix the $i^{th}$ row does <u>not</u> correspond in any way with job $J_i$, as was the case for the A and D matrices. In fact, the concept of a row in relation to matrix Q is somewhat misleading, since each column is an <u>independent</u> queue, with no necessary relationship between the items in corresponding row positions of different columns. It is assumed that the indices in a column are arranged such that the first $QSIZE_j$ elements of column j will be the indices of the jobs with unsatisfied demands for resource elements of type j, and the remaining $(n - QSIZE_j)$ elements will be undefined.

Since it is only the first $QSIZE_j$ elements that are of interest in what follows, conceptually it is possible to consider the length of each queue to be variable over time, with n the maximum possible length. The matrix notation is chosen here simply for convenience in formulating the algorithms, but in

some implementations it may be more efficient to represent Q as a set of ordered lists, one list for each resource class, and then the length of each list would vary dynamically with the corresponding value of $\overline{QSIZE}$.

The elements in column j of Q must be ordered according to increasing $D_{ij}$. This is an essential part of the fast deadlock detection algorithm and is discussed in section II. J.

Item 9: COUNT--a scalar variable whose value is the number of jobs $J_i$ with $DNUMB_i > 0$.

In terms of the function lambda defined previously, COUNT can be expressed precisely as:

$$COUNT(t) = n - \sum_{i=1}^{n} \lambda(i,t)$$

This obviously implies the following relationship for all times t:

$$0 \leq COUNT(t) \leq n \qquad \text{(II. F. 9)}$$

Item 10: $\overline{E}$--an n-component vector, called the "allocation sequence" vector. It represents an ordered list of indices of jobs $J_i$ with $DNUMB_i > 0$.

The number of indices in $\overline{E}$ is given by the value of COUNT, since a job index is entered in $\overline{E}$ only once for a REQUEST, regardless of the number of different resource classes affected by the REQUEST. As with the columns of Q, $\overline{E}$ is represented as a vector simply for convenience in formulating the algorithms which follow. In some implementations it might be better constructed as a variable length list whose length is given by COUNT. The order of indices in $\overline{E}$ represents the order in which jobs with unsatisfied demands are to be satisfied. In other words, $\overline{E}$ represents the schedule to be followed in

allocating resources when they become available. This allocation sequence is produced by the deadlock detection algorithms.

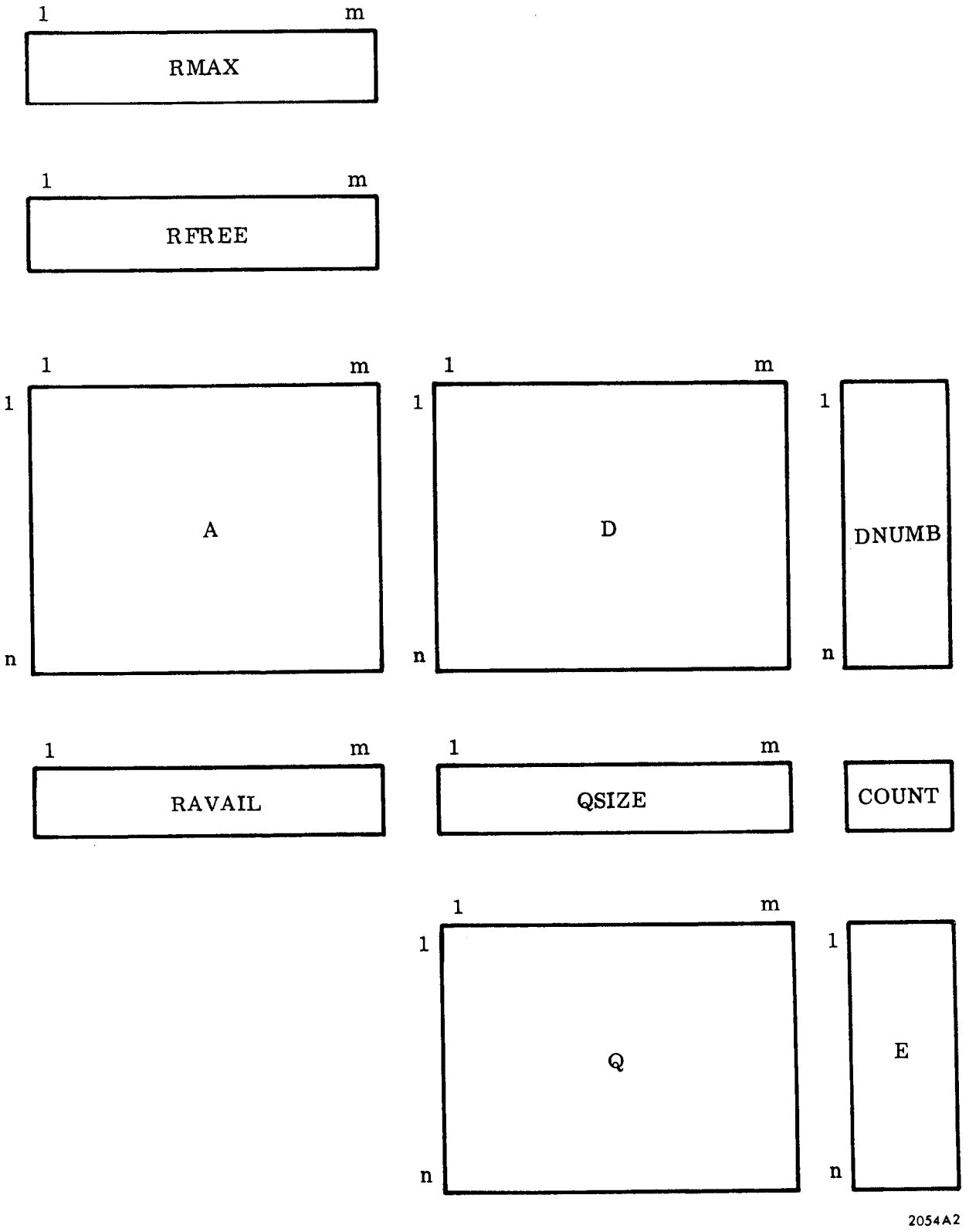


Figure II. F. 1 Scheduler Data Base $B_1$ for System $\hat{S}$.

G. Formal Properties of $\hat{S}$

Definition 1: (Finite Control) For each job $J_i$, if for every time t such that $DNUMB_i(t) > 0$ there exists a time $t' \geq t$ such that $DNUMB_i(t') = 0$, then after a finite number of such t and t' there exists a time t'' greater than all t and t' such that $A_{ij}(t'') = 0$ for $j = 1, 2, \ldots, m$.

This first definition is just a restatement of assumption 4 in terms of the data structures defined in $B_1$.

Definition 2: At time t job $J_i$ is in permanent wait if $DNUMB_i(t) > 0$, and for all time $t' \geq t$, $DNUMB_i(t') > 0$.

Definition 3: A system $\hat{S}$ is deadlocked at time t if and only if there exists at least one job $J_i$ in $\hat{J}$ that is in permanent wait.

Since much of what follows will be dealing with ordered sequences of the jobs in set $\hat{J}$, the following notion will be useful.

Definition 4: A set $\hat{J}'$ containing the n elements $J'_1, J'_2, \ldots, J'_n$ is a permutation of set $\hat{J}$ containing the n elements $J_1, J_2, \ldots, J_n$ if and only if for every i, $1 \leq i \leq n$ there exists a k, $1 \leq k \leq n$ such that:

(a) $J'_i = J_k$

(b) $J'_i \neq J_j$ for any $j \neq k$

Such a set $\hat{J}'$ will be called a permutation, or permutation sequence of set $\hat{J}$.

It will also be assumed that operations on vectors are just a notational convenience for indicating that an operation is to be applied to each component of the vector. Thus, if $\overline{W}$ and $\overline{V}$ are k-component vectors, and c is a scalar:

$\overline{W} > c$ is true if and only if $W_i > c$ for $i = 1, 2, \ldots, k$

$\overline{W} \leq \overline{V}$ is true if and only if $W_i \leq V_i$ for $i = 1, 2, \ldots, k$

$\overline{W} + \overline{V}$ means $W_i + V_i$ for $i = 1, 2, \ldots, k$.

Definition 5: A finishing sequence F(t) for system $\hat{S}$ at time t is a permutation sequence $\hat{J}' = (J'_1, J'_2, \ldots, J'_n)$ of set $\hat{J}$ such that associated with each job $J'_i$ is a time $t_i$ and an ordering of these times $t = t_1 \leq t_2 \leq \ldots \leq t_n < \infty$ such that for each i = 1, 2, ..., n the following is true:

$$DNUMB_i(t') > 0 \quad \text{for any } t' \text{ with } t \leq t' < t_i$$

$$DNUMB_i(t_i) = 0.$$

Theorem 1: A system $\hat{S}$ is not deadlocked at time t if and only if there exists at least one finishing sequence F of all jobs in $\hat{J}$.

Proof:

If: Existence of F implies that for each job $J_i$ in $\hat{J}$ there exists a time $t_i$ when $DNUMB_i(t_i) = 0$, by definition 5. Therefore, by definition 2, no job is in permanent wait, which by definition 3 implies that $\hat{S}$ is not deadlocked.

Only if: Suppose that $\hat{S}$ is not deadlocked. By definition 3, no job in $\hat{J}$ is in permanent wait. Therefore, by definition 2, for each job $J_i$ in $\hat{J}$ there must exist a time $t_i$, $t \leq t_i < \infty$ such that $DNUMB_i(t_i) = 0$. Let $t' = (t'_1, t'_2, \ldots, t'_n)$ be a permutation sequence of the set of times $t = (t_1, t_2, \ldots, t_n)$, arranged such that $t'_1 \leq t'_2 \leq \ldots \leq t'_n$. The set $\hat{J}'$ is then constructed as a permutation sequence of set $\hat{J}$ by arranging the jobs in $\hat{J}'$ in the same order as their associated times are ordered in t'. This set $\hat{J}'$ satisfies definition 5, by construction, and hence forms a finishing sequence for $\hat{S}$ at time t. Q. E. D.

In all the following theorems, it is assumed that F is a permutation sequence of all jobs in $\hat{J}$, with the jobs renumbered so that $J_k$ is the $k^{th}$ job in F.

Theorem 2: If F is a finishing sequence for system $\hat{S}$ at time t, then the following conditions are satisfied by jobs in F:

$$\overline{D}_1(t) \leq \overline{RFREE}(t)$$

$$\overline{D}_i(t) \leq RFREE(t) + \sum_{k=1}^{i-1} \overline{A}_k(t) \qquad i = 2, 3, \ldots, n \qquad \text{(II. G. 1)}$$

Proof (by induction):

Let F be a finishing sequence at time t. By definition 5, for the first job in F, $J_1$, we will have $\overline{D}_1(t_1) = 0$ at some time $t_1$, $t \leq t_1 \leq t_i$ for $i = 2, 3, \ldots, n$. Assume that condition (II. G. 1) does not hold for $J_1$ at time t. This implies that for at least one value of j, $D_{1j}(t) > RFREE_j(t) \geq 0$. Hence $\overline{D}_1(t)$ cannot be reduced to zero by an assignment of resource elements of type j until $RFREE_j$ increases. This happens only by a RELEASE invoked by some job $J_k$ currently allocated resource elements of type j, and $k \neq 1$ by consequence 4. However, only jobs $J_i$ with $\overline{D}_i = 0$ can invoke a RELEASE primitive, so that $J_k$ must have $\overline{D}_k(t_k) = 0$ at some time $t_k$, $t \leq t_k < t_1$, in order to be able to invoke RELEASE at time $t_k'$, $t_k < t_k' \leq t_1$. Therefore, at time $t_k$, $\overline{D}_k(t_k) = 0$ but $D_{1j}(t_k) > 0$ for at least one value of j, and since $t \leq t_k < t_1$, $J_k$ must precede $J_1$ in F, by definition 5. This contradicts the assumption that $J_1$ is the <u>first</u> job in F. Therefore, (II. G. 1) must hold for $J_1$ at time t. Next assume (II. G. 1) holds for jobs $J_1, J_2, \ldots, J_{i-1}$ in F. We will show that it also must hold for the next job in F, $J_1$. Assume that (II. G. 1) does not hold for $J_i$. By definition 5, all jobs $J_k$, with $i \leq k \leq n$, must have $\overline{D}_k(t') > 0$ for all t' such that $t \leq t' \leq t_{i-1}$. Since none of the resource elements assigned to these jobs can be released during this time, by consequence 4, the maximum value of $\overline{RFREE}$ at time $t_{i-1}$ is

$\overline{RMAX} - \sum_{k=i}^{n} \overline{A}_k(t_{i-1}) = \overline{RMAX} - \sum_{k=i}^{n} \overline{A}_k(t)$. This maximum can be achieved only if all jobs $J_p$, with $p < i$, have released control of all their resource elements by time $t_{i-1}$. Using relation (II. F. 5) to substitute for $\overline{RMAX}$, the above limit on $\overline{RFREE}$ becomes $\overline{RFREE}(t_i) \leq \overline{RFREE}(t) + \sum_{k=1}^{i-1} \overline{A}_k(t)$, and since we assume (II. G. 1) does not hold for $J_i$, this gives us $\overline{RFREE}(t_i) < \overline{D}_i(t_i)$. Therefore even if all jobs $J_p$ with $p < i$ have released control of all their resources by time $t_{i-1}$, an assignment to $J_i$ still cannot occur at time $t_i$ since sufficient free resources do not exist. Thus we cannot achieve $\overline{D}_i(t_i) = 0$ for any $t_i \geq t_{i-1}$ until some other job $J_k$ with $k > i$ RELEASEs some of its resource elements. But this job $J_k$ would have to have $\overline{D}_k(t_k) = 0$ at some time $t_k < t_i$ in order to be able to invoke RELEASE at time t', with $t_k < t' \leq t_i$. Therefore at time $t_k$, we have for at least one value of j, $D_{ij}(t_k) > 0$, but $\overline{D}_k(t_k) = 0$, and since $t_k < t_i$, $J_k$ must precede $J_i$ in sequence F, by definition 5. However, this contradicts the assumption that $J_i$ is the next job in F. Therefore (II. G. 1) must also hold for $J_i$, which proves the induction. Q. E. D.

Theorem 2 gives a necessary condition that must be satisfied by all finishing sequences of $\hat{S}$. Therefore, if at time t there does not exist any permutation sequence of $\hat{J}$ satisfying conditions (II. G. 1), it is obvious that there cannot exist a finishing sequence, and hence $\hat{S}$ is deadlocked, by theorem 1. This is stated formally in the next theorem.

Theorem 3: If at time t there does not exist a permutation sequence F of all jobs in $\hat{J}$ that satisfies conditions (II. G. 1), system $\hat{S}$ is deadlocked at time t.

But what if such a sequence does exist--is it still possible that $\hat{S}$ is deadlocked? In other words, is the existence of a permutation sequence of $\hat{J}$ satisfying (II. G. 1) also a sufficient condition for $\hat{S}$ to be not deadlocked? The answer is that without advance information about the future resource requirements of the jobs in $\hat{J}$, it is impossible to say. Even though it may be possible to arrange the jobs into a permutation sequence satisfying (II. G. 1), there is no way to guarantee that job demands will be satisfied in that order. Active jobs may make new requests that will force a rearrangement of the allocation order predicted by F. New jobs may be created and existing jobs may terminate, so that F will no longer be a permutation sequence for $\hat{J}$.

However, it is conceivable that no new demands will be made, or that they will not destroy the order predicted by F; that no new jobs will be created; and that no jobs will terminate, or that they will terminate in the order predicted by F. Therefore F could, in fact, turn out to be a valid finishing sequence, but at time t there is no way of accurately predicting this without advance information on job behavior. At least deadlock is not inevitable, since a possible finishing sequence does exist. Given the present state of a system, and the lack of any advance information on job behavior, existence of a permutation sequence of all jobs in $\hat{J}$ satisfying (II. G. 1) implies that deadlock cannot be detected at this time. Only future actions of the jobs themselves will create or ascertain a deadlock situation.

The next theorem shows that by making a simple assumption on the future resource requirements of jobs in $\hat{J}$, it is possible to take one of the permutation sequences satisfying (II. G. 1) at time t and allocate resources to jobs in the order that they appear in the sequence, with the result that all current resource

requests will be satisfied in a finite time. A finishing sequence is thereby produced, so that $\hat{S}$ is not deadlocked at time t under this assumption.

Theorem 4: Without advance information, any permutation sequence F of all jobs in $\hat{J}$ that satisfies (II. G. 1) at time t is a finishing sequence for system $\hat{S}$ under the assumption that no new resource REQUESTs will be made at any future time $t' > t$.

Proof (by induction):

$J_1$ is the first job in F, so that $\overline{D}_1(t) \leq \overline{RFREE}(t)$, by (II. G. 1). Therefore, at time $t_0 \geq t$, if $\overline{D}_1(t) > 0$, it is possible to make an assignment to $J_1$ that will make $\overline{D}_1(t_0) = 0$, $\overline{A}_1(t_0) = \overline{A}_1(t) + \overline{D}_1(t)$, and reduce $\overline{RFREE}(t_0)$ to $\overline{RFREE}(t) - \overline{D}_1(t)$. (If $\overline{D}_1(t) = 0$, no assignment to $J_1$ is necessary, since $t_0 = t$, $\overline{RFREE}(t_0) = \overline{RFREE}(t) - \overline{D}_1(t)$, and $\overline{A}_1(t_0) = \overline{A}_1(t) + \overline{D}_1(t)$ trivially.) By assumption no job in $\hat{J}$ will make any new REQUEST after time t, so that by definition 1, $J_1$ will RELEASE all its resources at some finite time $t_1 > t_0$, thereby returning all the resources $\overline{A}_1(t_0)$ to the free state. Hence $\overline{RFREE}(t_1) = \overline{RFREE}(t_0) + \overline{A}_1(t_0) = \overline{RFREE}(t) - \overline{D}_1(t) + \overline{A}_1(t) + \overline{D}_1(t) = \overline{RFREE}(t) + \overline{A}_1(t)$.

Now assume that jobs $J_1, J_2, \ldots, J_{i-1}$ have had their demands satisfied and have released their resources by time $t_{i-1}$, and that

$$\overline{RFREE}(t_{i-1}) = \overline{RFREE}(t) + \sum_{k=1}^{i-1} \overline{A}_k(t). \qquad \text{(II. G. 2)}$$

We will show that at some time $t_i \geq t_{i-1}$, job $J_i$ will also achieve this state and (II. G. 2) will hold for i. By condition (II. G. 1),

$$\overline{D}_i(t) \leq \overline{RFREE}(t) + \sum_{k=1}^{i-1} \overline{A}_k(t) = \overline{RFREE}(t_{i-1}).$$

Since assignments have been made only to jobs $J_p$ with $p < i$, $\overline{D}_i(t) = \overline{D}_i(t_{i-1})$ and $\overline{A}_i(t) = \overline{A}_i(t_{i-1})$. Therefore time $t'$, $t' \geq t_{i-1}$, if $D_{ij}(t) > 0$ for at least one value of j, it is possible to make an assignment to $J_i$ that will make $\overline{D}_i(t') = 0$, $\overline{A}_i(t') = \overline{A}_i(t) + \overline{D}_i(t)$, and $\overline{RFREE}(t') = \overline{RFREE}(t_{i-1}) - \overline{D}_i(t_{i-1})$ $= \overline{RFREE}(t) + \sum_{k=1}^{i-1} \overline{A}_k(t) - \overline{D}_i(t)$. (If $\overline{D}_i(t) = 0$, then these values of $\overline{D}_i(t')$, $\overline{A}_i(t')$, and $\overline{RFREE}(t')$ are obviously valid without an assignment.) By assumption $J_i$ will not make any new REQUEST after time t, so that by definition 1, $J_i$ will RELEASE all its resources by some finite time $t_i \geq t'$, thereby returning all $\overline{A}_i(t')$ resources to the free state. This gives:

$$\overline{RFREE}(t_i) = \overline{RFREE}(t') + \overline{A}_i(t') = \overline{RFREE}(t) + \sum_{k=1}^{i-1} \overline{A}_k(t) - \overline{D}_i(t)$$

$$+ \overline{A}_i(t) + \overline{D}_i(t)$$

$$= \overline{RFREE}(t) + \sum_{k=1}^{i} \overline{A}_k(t)$$

which proves the induction. Q. E. D.

Theorem 5: If there exists a permutation sequence F of all jobs in $\hat{J}$ that satisfies (II. G. 1) at time t, then it is impossible to detect a deadlock for system $\hat{S}$ at time t without further information on the resource requirements of jobs in $\hat{J}$. A system $\hat{S}$ for which such a sequence exists is said to be deadlock free at time t.

Proof:

Existence of such an F means that deadlock is not inevitable at time t, since by theorem 4 there is an assumption about future job behavior which is sufficient for this F to be a finishing sequence for $\hat{S}$. To disprove this

assumption requires knowing at time t that some job in $\hat{J}$ will in fact make a request for additional resource elements at some time $t' > t$. Without this information on future resource requirements, it cannot be shown that F is not a finishing sequence for $\hat{S}$ at time t, and hence it cannot be shown that $\hat{S}$ is deadlocked at time t, by theorem 1. Q. E. D.

## H. Scheduler Primitives

### 1) Job Primitives

The definition of REQUEST and RELEASE, the two job primitives at the highest hierarchical level in system $\hat{S}$, is given in figure II.H.1. As stated in section II.C, these are the only primitives which can be invoked directly by jobs, and obviously a job can invoke only one primitive in this pair at any instant of time. There exists the possibility for several jobs to execute REQUEST and/or RELEASE simultaneously, which necessitates the introduction of the critical section procedure 'STARTUP' that is invoked as part of the RELEASE primitive. This procedure constitutes a critical section as defined by Dijkstra [8], so that it can be executed by at most one job at any instant of time. The need for this is explained below.

The REQUEST primitive is extremely simple: it invokes the BLOCK primitive to cause the job making the REQUEST to enter the waiting state, and then invokes the ASSIGN primitive on behalf of this job. The decision as to whether or not the assignment can be made, and the bookkeeping associated with the outcome of this decision, are clearly left up to the ASSIGN primitive which is on the next level in the hierarchy below the job primitives. Control is not returned from the ASSIGN to the REQUEST primitive until the request has been satisfied by the actions of the ASSIGN. At that time, the REQUEST is completed by invoking the UNBLOCK primitive to permit the job to proceed in the active state once again.

The RELEASE primitive is only slightly more complicated than the REQUEST. It first invokes BLOCK to cause the job invoking the RELEASE to enter the waiting state, then it invokes the UNASSIGN primitive on behalf of this job. The UNASSIGN must decide when the resource elements can be released, and only

after this has been accomplished will control be returned from UNASSIGN to RELEASE. At this point, the procedure STARTUP is called.

Procedure STARTUP is needed for the following reason. Resources have been returned to the free state by the UNASSIGN which is completed just prior to the call to STARTUP, and which causes an increase in the number of resource elements that exist in the free state. It is possible that after such an increase, there would be a sufficient number of free resources to satisfy the resource demand of one of the jobs that is "asleep" on the $\overline{E}$ list. These jobs were put onto this list by PUTASLEEP at the time they invoked the REQUEST primitive because an insufficient number of resource elements of the correct types were free to satisfy the demand at that time. The object of STARTUP is simply to reinvoke the ASSIGN primitive on behalf of all the sleeping jobs in $\overline{E}$, so that ASSIGN can determine whether or not their demands can now be satisfied, and if so, to make the assignment and get the job awake again. This procedure must be a critical section so that a job in the waiting state will not be reactivated more than once for the same request, a situation that could arise if STARTUP were being executed simultaneously by two or more jobs. Upon completion of the critical section STARTUP, the RELEASE primitive completes its execution by invoking UNBLOCK, thereby permitting the job to proceed.

The job primitives contain no reference to deadlock detection and/or prevention, or any other scheduling constraints, since these functions are embodied in lower level primitives that constitute the true scheduling aspects of any system. The job primitives can be considered as simply the interface between the users of the scheduling facilities and these facilities themselves.

2) Intermediate Level Primitives

The definition of the two intermediate level primitives, ASSIGN and UNASSIGN, is given in figure II. H. 2. As stated in section II. C, primitive pairs at this level are constructed so that only one of the primitives in the pair can be executing at any instant of time. Thus the pair forms a critical section of the same type as was mentioned above. This is clearly necessary, since these primitives modify data structures in data base B that relate to the global state of the system rather than to the state of a single job, as was the case for the job primitives. It is essential that these modifications be carried out in an integral sequence in order to have well defined results and proper system behavior.

The actions of the ASSIGN primitive are quite straightforward. If insufficient resource elements are in the free state to satisfy the demand $\overline{N}$, the procedure PUTASLEEP is called to enter the job onto list $\overline{E}$, the list of all jobs with unsatisfied demands, and to perform any additional bookkeeping necessitated by the scheduler's operating constraints. If sufficient free resources do exist, the demand is immediately satisfied. In this case, the first step is to call the procedure GETAWAKE in order to perform any bookkeeping for the scheduler constraints (such as deadlock detection), and to remove the job from the list $\overline{E}$ if it was put there previously by PUTASLEEP. The next step is to increase the vector $\overline{A}$ for this job and decrease the vector $\overline{RFREE}$ by the amount $\overline{N}$, and then invoke the ALLOCATE primitive for each resource element demanded in order to select a free element of the proper type and establish access between it and the job. The details of this selection and the method of establishing control are contained in the ALLOCATE primitive, and are of no interest in this model.

The procedures PUTASLEEP and GETAWAKE, defined in figure II. H. 3, are really part of the ASSIGN primitive (they are called nowhere else), but have been written as auxiliary functions for two reasons: to demonstrate the symmetry in the definitions of ASSIGN and UNASSIGN; and to isolate the extra bookkeeping needed for the deadlock detection algorithms, thereby demonstrating its symmetry and simplifying the analysis of the costs of deadlock detection (see section II. K). This also permits the definition of the ASSIGN primitive to become independent of the operating constraints imposed by the scheduler, and to retain the same definition for a large class of systems that are describable by the model, whether or not they detect deadlock dynamically. (It must be shown that this ASSIGN is in fact correct for a scheduler that does attempt to detect deadlock dynamically--see theorem 6.)

The UNASSIGN primitive is the complement of the ASSIGN primitive as far as most of the bookkeeping is concerned, but is considerably simpler because the decision making functions are eliminated. (It is proved in theorem 7 that resources can always be unassigned at any time without creating a deadlock.) Therefore the UNASSIGN simply decreases the $\overline{A}$ vector for the job and increases the $\overline{RFREE}$ vector by the number of resource elements being released, and then invokes the DEALLOCATE primitive for each of these elements to remove the access between that element and the job. This will also return the element to the free state.

In the definition of UNASSIGN we have introduced the notation $\delta(N)$, where N is a set of resource elements, to mean the m-component vector, called the "measure" of set N, whose $j^{th}$ component $\delta(N)_j$ is the number of resource elements of type j in set N.

3) Formal Properties of the Primitives

The ASSIGN primitive defined in figure II. H. 2 satisfies a demand $\overline{N}$ from job $J_i$ at time t if and only if $N \leq RFREE(t)$. This is done independently of any deadlock detection scheme (which has yet to be introduced). The next theorem proves that this is in fact the correct decision mechanism for the ASSIGN primitive, since when the condition

$$0 < \overline{N} \leq \overline{RFREE}(t) \qquad \text{(II. H. 1)}$$

is true, an assignment to $J_i$ cannot create a deadlock.

Theorem 6 is stated in terms of demand $\overline{D}_i$ for job $J_i$, to be consistent with the other formal results. However, in the definition of the primitive ASSIGN the demand is represented by parameter $\overline{N}$, both as a matter of convenience and to demonstrate that if the demand can be satisfied at the time ASSIGN is invoked by REQUEST, the extra bookkeeping to copy the value of $\overline{N}$ into $\overline{D}_i$ is not necessary, since $\overline{D}_i$ will be immediately reset to zero. The copying and resetting are contained in the procedures PUTASLEEP and GETAWAKE respectively to indicate that it is necessary in an operational situation only when a job is being put asleep or gotten awake. Conceptually parameter $\overline{N}$ is identical to vector $\overline{D}_i$, both representing the unsatisfied demand of job $J_i$.

Theorem 6: If $\overline{D}_i(t) \leq \overline{RFREE}(t)$ for some job $J_i$ at time t and $\hat{S}$ is deadlock-free at time t, then $\hat{S}$ will remain deadlock-free after an ASSIGN to $J_i$ at time t.

Proof:

Let F be a sequence of all jobs in $\hat{J}$ that satisfies (II. G. 1) at time t (there must be at least one if $\hat{S}$ is deadlock-free, by theorem 5), and let the jobs be renumbered so that $J_k$ is the $k^{th}$ job in F. Suppose the assignment is

made to $J_i$ at time t. Consider the state of the system at time t' immediately after the assignment is complete:

$$\overline{RFREE}(t') = \overline{RFREE}(t) - \overline{D}_i(t) \geq 0$$

$$\overline{D}_i(t') = 0$$

$$\overline{A}_i(t') = \overline{A}_i(t) + \overline{D}_i(t) \geq 0$$

Consider the sequence F' constructed from F by removing $J_i$ from its position in F and making it the first job in F', so that $F' = \{J_i, J_1, J_2, \ldots, J_{i-1}, J_{i+1}, J_{i+2}, \ldots, J_n\}$. Then F' will satisfy (II. G. 1) for system $\hat{S}$ at time t', as is shown next.

Obviously $0 = \overline{D}_i(t') \leq \overline{RFREE}(t')$, so that $J_i$ satisfies (II. G. 1) as the first job in sequence F'. For any job $J_k$ with $k < i$, we have:

$$\overline{D}_k(t') = \overline{D}_k(t) \leq \overline{RFREE}(t) + \sum_{p=1}^{k-1} \overline{A}_p(t)$$

$$= \overline{RFREE}(t') + \overline{D}_i(t) + \sum_{p=1}^{k-1} \overline{A}_p(t')$$

$$= \overline{RFREE}(t') + \overline{A}_i(t') - \overline{A}_i(t) + \sum_{p=1}^{k-1} \overline{A}_p(t')$$

$$= \overline{RFREE}(t') + \overline{A}_i(t') + \sum_{p=1}^{k-1} \overline{A}_p(t')$$

where we have used the fact that $\overline{A}_p$ is always non-negative and that for $p \neq i$, $\overline{A}_p(t') = \overline{A}_p(t)$. Therefore these jobs also satisfy (II. G. 1) in the order in which they appear in F'. Next consider any job $J_k$ with $k > i$.

We have then:

$$\overline{D}_k(t') = \overline{D}_k(t) \leq \overline{RFREE}(t) + \sum_{p=1}^{k-1} \overline{A}_p(t)$$

$$= \overline{RFREE}(t) + \sum_{p=1}^{i-1} \overline{A}_p(t) + \overline{A}_i(t) + \sum_{p=i+1}^{k-1} \overline{A}_p(t)$$

$$= \overline{RFREE}(t') + \overline{D}_i(t) + \sum_{p=1}^{i-1} \overline{A}_p(t') + \overline{A}_i(t') - \overline{D}_i(t) + \sum_{p=i+1}^{k-1} \overline{A}_p(t')$$

$$= \overline{RFREE}(t') + \sum_{p=1}^{k-1} \overline{A}_p(t').$$

Therefore, (II. G. 1) holds for these jobs as well, which implies that $\hat{S}$ is deadlock-free at time t', by theorem 5. Q. E. D.

The next theorem proves that the UNASSIGN primitive can always perform a deallocation of resources without creating a deadlock. Therefore, there is no need for a decision mechanism in the UNASSIGN primitive as there was in the ASSIGN, and since there is no possibility of the job invoking the RELEASE to be delayed until the number of free resources is changed by some other job, the elaborate PUTASLEEP and GETAWAKE mechanism is also unnecessary.

Theorem 7: If job $J_i$ issues a RELEASE at time t and $\hat{S}$ is deadlock-free at time t, then $\hat{S}$ will remain deadlock-free after an UNASSIGN from $J_i$ at time t.

Proof:

Let F be a sequence of all jobs in $\hat{J}$ satisfying (II. G. 1) at time t (if $\hat{S}$ is deadlock-free at time t there must be at least one such F, by theorem 5), with the jobs renumbered so that $J_k$ is the $k^{th}$ job in F. Assume the

UNASSIGN is made at time t, so that at time t' immediately after the UNASSIGN is complete:

$$\overline{RFREE}(t') = \overline{RFREE}(t) + \delta(N)$$

$$\overline{D}_i(t') = \overline{D}_i(t) = 0$$

$$\overline{A}_i(t') = \overline{A}_i(t) - \delta(N)$$

and $\overline{A}_k(t') = \overline{A}_k(t)$, $\overline{D}_k(t') = \overline{D}_k(t)$ for all $k \neq i$.

Then F will still satisfy (II. G. 1) at time t', as is shown next. For any job $J_i$ with $k \leq i$, since F satisfies (II. G. 1) at time t:

$$\overline{D}_k(t') = \overline{D}_k(t) \leq \overline{RFREE}(t) + \sum_{p=1}^{k-1} \overline{A}_p(t)$$

$$= \overline{RFREE}(t') - \delta(N) + \sum_{p=1}^{k-1} \overline{A}_p(t')$$

$$= \overline{RFREE}(t') + \sum_{p=1}^{k-1} \overline{A}_p(t')$$

Therefore job $J_k$ with $k \leq i$ satisfies (II. G. 1) at time t'. Next consider any job $J_k$ with $k > i$. We then have:

$$\overline{D}_k(t') = \overline{D}_k(t) \leq \overline{RFREE}(t) + \sum_{p=1}^{k-1} \overline{A}_p(t)$$

$$= \overline{RFREE}(t) + \sum_{p=1}^{i-1} \overline{A}_p(t) + \overline{A}_i(t) + \sum_{p=i+1}^{k-1} \overline{A}_p(t)$$

$$= \overline{RFREE}(t') - \delta(N) + \sum_{p=1}^{i-1} \overline{A}_p(t') + \overline{A}_i(t') + \delta(N) + \sum_{p=i+1}^{k-1} \overline{A}_p(t')$$

$$= \overline{RFREE}(t') + \sum_{p=1}^{k-1} \overline{A}_p(t').$$

Therefore (II. G. 1) holds for these jobs at time t' also, which implies that F satisfies (II. G. 1) at time t', and hence $\hat{S}$ is deadlock-free at time t', by theorem 5. Q. E. D.

```
REQUEST(N̄)          invoked by job J_i
    begin
    BLOCK(J_i);
    ASSIGN(i, N̄);
    UNBLOCK(J_i);
    end

RELEASE(N)          invoked by job J_i
    begin
    BLOCK(J_i);
    UNASSIGN(i, N);
    STARTUP;
    UNBLOCK(J_i);
    end

critical section procedure STARTUP;
    begin
    parallel for each k in Ē do ASSIGN(k, D̄_k);
    end
```

Figure II. H. 1 The Job Primitives in System $\hat{S}$

```
ASSIGN(k, N̄)
    if N̄ ≤ RFREE
    then begin
        GETAWAKE(k, N̄);
        parallel for j in [1,m] do
S1:         begin
            A_kj +:= N_j;
            RFREE_j -:= N_j;
            for p from 1 step 1 up to N_j do ALLOCATE(J_k, j);
            end;
        end
    else PUTASLEEP(k, N̄);

UNASSIGN(k, N)
    parallel for j in [1,m] do
U1:     begin
        A_kj -:= δ(N)_j;
        RFREE_j +:= δ(N)_j;
        for each r_i of type j in N do DEALLOCATE(J_k, r_i);
        end;
```

Figure II. H. 2 The Intermediate Level Primitives for System $\hat{S}$

```
    procedure PUTASLEEP(k, N̄);
        if k not in Ē then
P1:         begin
            parallel for j in [1,m] do
P2:             begin
                RAVAIL_j -:= A_kj;
                if N_j > 0 then
                    begin
                    QSIZE_j +:= 1;
                    ENQUEUE(J_k, Q̄_j);
                    DNUMB_k +:= 1;
                    D_kj +:= N_j;
                    end;
                end;
            COUNT +:= 1;
            ENQUEUE(J_k, Ē);
            end;
```

Figure II. H. 3 Auxiliary Scheduling Functions for Deadlock Detection Bookkeeping

```
procedure GETAWAKE(k, N̄);
    if k in Ē then
G1:     begin
        parallel for j in [1,m] do
G2:         begin
            RAVAIL +:= A_kj;
            if N_j > 0 then
                begin
                QSIZE_j -:= 1;
                DEQUEUE(J_k, Q̄_j);
                DNUMB_k -:= 1;
                D_kj -:= N_j;
                end;
            end;
        COUNT -:= 1;
        DEQUEUE(J_k, Ē);
        end;
```

Figure II. H. 3 (continued)

## I. Incorporating Deadlock Detection Algorithms

The previous section demonstrated that an UNASSIGN can never create a deadlock, and neither can an ASSIGN when the condition (II. H. 1) is true. This implies that a deadlock can only occur when this condition does not hold; in other words, when:

$$N_j > RFREE_j(t) \qquad \text{for at least one } j. \qquad \text{(II. I. 1)}$$

This is in fact correct for jobs which are just making new REQUESTs that are unsatisfiable at the time they are made. This seems reasonable, since it is at this time that a job is entered onto the list $\overline{E}$ (the wait list) by the procedure PUTASLEEP, where it must remain until some other job releases sufficient resources to get this job awake again. During the time a job is "asleep" on the list $\overline{E}$, the resources already allocated to it are "frozen", as stated in consequence 5, and by consequence 4, there is no way control of these elements can be removed from the sleeping job until the job becomes active. Frozen resources are always owned, and must remain owned at least until the job owning them is awakened. Clearly there is a danger that the number of resources frozen in this way will be very large, and the situation may arise where even if all the unfrozen resource elements were in the free state, there would still be an insufficient number to satisfy the demand of any sleeping job. Hence these jobs would "sleep forever", i.e., they would be in permanent wait and hence the system $\hat{S}$ is deadlocked.

With this in mind, it is easy to see how to formulate an efficient method of deadlock detection. We assume that in its initial state, system $\hat{S}$ is not deadlocked. Since no RELEASE, and no REQUEST satisfying (II. H. 1) can cause a deadlock, only REQUESTs with (II. I. 1) true will require a check to see if the

critical threshold of frozen resources will be passed when this job is put asleep. Clearly this check only has to be made when a job is first put asleep, since it is at this time that the resources become frozen. If at some later time an ASSIGN on behalf of this job is invoked by STARTUP, and the condition (II. H. 1) still does not hold, no additional check is necessary because no new deadlock can be created at that time. Therefore the checking algorithm need be executed only once, at the time a job is first put asleep, which is when it is entered onto the $\overline{E}$ list. In fact, a very simple way to integrate a deadlock detection algorithm into the definitions given so far is simply to make it the procedure which enqueues a new item onto the $\overline{E}$ list (i.e., ENQUEUE($J_k$, $\overline{E}$)).

The reasoning for this is quite straightforward. The list $\overline{E}$ is defined as the order to be followed in satisfying the demands of sleeping jobs. The algorithm for deadlock detection, formulated in the next section, constructs a sequence of the jobs in $\hat{J}$ that satisfies (II. G. 1). By theorem 4, this sequence is a potential finishing sequence for system $\hat{S}$ at time t, which implies that it defines an order in which job demands can be satisfied without causing a deadlock. If such an ordering does not exist, the system is deadlocked, as was shown in theorem 3.

It should be clear that in such a sequence, if k jobs are active at time t, the first k jobs in that sequence can be these jobs, since $\overline{D}_i(t) = 0$ for $i = 1, 2, \ldots, k$, and (II. G. 1) is satisfied trivially, regardless of the ordering between these jobs in the sequence. Therefore an algorithm to check for the existence of a sequence satisfying (II. G. 1) can always assume that the first k jobs in the sequence will be the k jobs $J_i$ with $\overline{D}_i(t) = 0$. Therefore, if the initialization for the algorithm is done properly (which is the function of the bookkeeping on the RAVAIL vector), a deadlock detection algorithm need only be concerned with the ordering of the elements in $\overline{E}$ that will make this list,

prefaced by the list of active jobs in arbitrary order, a sequence satisfying (II. G. 1).

Therefore we simply define the ENQUEUE procedure for list $\overline{E}$ to be the deadlock detection algorithm. This algorithm will be formulated to attempt to order the jobs in $\overline{E}$ to satisfy conditions (II. G. 1) (when prefaced by the active jobs). If it succeeds, no deadlock is detectable at that time. If no such ordering exists, a deadlock has been created by the job being enqueued.

## J. Algorithm $L_1$

The first deadlock detection algorithm to be considered is given in figure II. J. 1, with the critical section procedure needed by $L_1$ given in figure II. J. 2. The algorithm requires the following data structures for use as temporary storage:

$\overline{\text{MARK}}$--an m-component vector associated with the Q matrix; the $j^{th}$ component is the index of the "next" item in column j of Q to be used in $L_1$.

$\overline{\text{THRESH}}$--an m-component vector associated with $\overline{\text{RAVAIL}}$ and A; its value is defined below.

$\overline{\text{DN}}$--an n-component vector associated with $\overline{\text{DNUMB}}$; its value is defined below.

N--a scalar associated with the list $\overline{E}$.

OLDN--a scalar whose value is the value of N at the start of the current iteration of $L_1$.

The intuitive idea behind the algorithm $L_1$ is quite simple. Our objective is to order the jobs in $\overline{E}$ so that conditions (II. G. 1) are satisfied. One possible way of doing this is to build up the sequence in $\overline{E}$ by repeatedly searching the entire set of waiting jobs not already ordered in $\overline{E}$ for a next one which satisfies (II. G. 1). This is the method proposed by Habermann [12], and is guaranteed to find a sequence satisfying (II. G. 1) if one exists. However, if there are n jobs, this algorithm requires up to (n-1) searches, the first search requiring up to (n-1) comparisons, the second search (n-2) comparisons, etc., giving an algorithm that, in the worst case, requires on the order of $n^2/2$ operations to order $\overline{E}$. (In many practical cases, this extreme limit can be avoided, see [11].) If we can eliminate the repeated searching by always having the "next" job ready

to test in condition (II. G. 1), the work involved will be reduced to at most n operations in the worst case, a significant improvement.

The function of the Q matrix is simply to have the "next" job ready to test in algorithm $L_1$. Each column j of Q contains the indices i of jobs $J_i$ with $D_{ij} > 0$ (i.e., with unsatisfied demands for resources of type j), and these indices are ordered according to increasing size of the demand $D_{ij}$.

Definition 6: The ordering rule in system $\hat{S}$ for elements in each column of Q is such that if i precedes k in column j of Q, then:

$$0 < D_{ij} \leq D_{kj} \qquad \text{(II. J. 1)}$$

for all such i and k in column j.

Each column of Q is independent of all other columns, and this ordering must be produced by the lowest level primitive ENQUEUE when applied to columns of Q (i.e., ENQUEUE($J_k$, $\overline{Q}_j$)).

Algorithm $L_1$ is an iterative procedure, with at most n iterations, constructed to order as many jobs as possible in sequence $\overline{E}$ on each iteration. Beginning at the top of each column j of Q on the first iteration, and on succeeding iterations picking up where the previous iteration left off, the algorithm tests the next item in the column, say i, to see if:

$$D_{ij} \leq \text{THRESH}_j. \qquad \text{(II. J. 2)}$$

If so, job $J_i$ is "approved" and testing continues with the next item in the $j^{th}$ column. If there is no next item, or if (II. J. 2) does not hold for the current index i, the iteration for this value of j terminates. The $j^{th}$ component of the m-component vector $\overline{\text{MARK}}$ is used to "remember" which is the "next" item in column j of Q at each step. At the beginning of each iteration, the vector

$\overline{\text{THRESH}}$ will have as its value:

$$\overline{\text{THRESH}} = \overline{\text{RFREE}} + \sum_{p=1}^{k} \overline{A}_p \qquad \text{(II. J. 3)}$$

where k is the number of jobs "completely approved" at the start of this iteration. In the initialization of $L_1$, $\overline{\text{THRESH}}$ is set to $\overline{\text{RAVAIL}}$, so that on the first iteration k is simply the number of jobs $J_i$ with $\overline{\text{DNUMB}}_i = 0$, and $\overline{\text{THRESH}}$ represents the number of "unfrozen" resource elements in the system at that instant. Job $J_i$ is "completely approved" only after it has been separately "approved" once for each component of $\overline{D}_i$ greater than zero, which means a total of $\text{DNUMB}_i$ separate approvals for each waiting job $J_i$. When a job becomes "completely approved", it is added to $\overline{E}$ as the next job in a sequence satisfying (II. G. 1). At the end of an iteration, if all waiting jobs have been ordered in $\overline{E}$, the algorithm is finished and no deadlock is detected. If no jobs were added to $\overline{E}$ in this iteration, the algorithm is "stuck", since there is no next job satisfying (II. G. 1) and as shown in theorem 9 this means that there exists no sequence satisfying (II. G. 1) for system $\hat{S}$ at this time, which by theorem 3 means that $\hat{S}$ is deadlocked. The key concept in being able to show that this is true is that, due to the ordering of the columns of Q, if condition (II. J. 2) does not hold on any iteration for job $J_i$, then for any k following i in the $j^{th}$ column of Q,

$$D_{kj} \geq D_{ij} > \text{THRESH}_j. \qquad \text{(II. J. 4)}$$

This means that if job $J_i$ fails the test (II. J. 2), all other jobs following $J_i$ in column j of Q are also known to fail the test without having to be checked.

On the other hand if jobs were added to $\overline{E}$ on an iteration, but not all jobs are ordered in $\overline{E}$ at the end of that iteration, the resources $\overline{A}$ controlled by the

jobs added to $\overline{E}$ on that iteration are added to $\overline{THRESH}$, so that, when the next iteration is begun, its value will be given by (II. J. 3).

The next theorems prove formally that this algorithm uniquely determines whether or not a sequence satisfying (II. G. 1) exists for system $\hat{S}$ at time t. The proof of theorem 8 is by induction, showing simply that at the beginning of each execution of the loop labelled REPEAT in figure II. J. 1, the value of $\overline{THRESH}$ is given by (II. J. 3), and that the jobs as ordered in $\overline{E}$, preceded by the active jobs in any order, satisfy (II. G. 1). Theorem 9 shows that if this algorithm fails to add a new job to $\overline{E}$ on any iteration, then no sequence satisfying (II. G. 1) exists. This proof is based on the ordering of the elements in the columns of Q.

Theorem 8: A permutation sequence F of all jobs in $\hat{J}$ that is composed of all jobs with $\overline{D}_p = 0$, followed by the list $\overline{E}$ generated by algorithm $L_1$, will satisfy (II. G. 1).

Proof (by induction):

Assume that the jobs are renumbered so that the k jobs $J_p$ with $\overline{D}_p = 0$ have indices less than or equal to k ($p \leq k$). Since $0 \leq \overline{RFREE}$ (by II. F. 2), obviously:

$$\overline{D}_p = 0 \leq \overline{RFREE} \qquad \text{for } p = 1, 2, \ldots, k.$$

Since $\overline{A}_i \geq 0$ for all i (by II. F. 3), it is also obvious that

$$\overline{D}_p \leq RFREE + \sum_{i=1}^{p-1} \overline{A}_i \qquad \text{for } p = 2, 3, \ldots, k,$$

which means that (II. G. 1) is satisfied by the first k jobs in F, regardless of their order in F.

In the initialization phase of $L_1$, $\overline{THRESH}$ is set to the value of $\overline{RAVAIL}$, which is defined in item 6 to be $\overline{RFREE} + \sum_{i=1}^{k} \overline{A}_i$, since only jobs $J_i$ with $i \leq k$ have $\overline{D}_i = 0$ and hence $DNUMB_i = 0$. The vector $\overline{DN}$ is initialized to the value of $\overline{DNUMB}$, which is defined in item 5 such that the $i^{th}$ component is the number of resource classes $R_j$ for which $D_{ij} > 0$. Scalars N and OLDN are initialized to zero, and the vector $\overline{MARK}$, whose $j^{th}$ component is a pointer to the "next" element in the $j^{th}$ column of Q, is initialized to one to indicate that initially the next item in each column is in fact the first item. The inductive assumption is that at the start of each execution of the statement labelled REPEAT in figure III. J. 1, the sequence of jobs in F, consisting of the k jobs $J_i$ with $DNUMB_i = 0$ followed by the N jobs $J_i$ with $DNUMB_i > 0$ so far ordered in $\overline{E}$ by algorithm $L_1$, satisfies (II. G. 1), and the value of THRESH is:

$$\overline{THRESH} = \overline{RFREE} + \sum_{i=1}^{k+N} \overline{A}_i. \qquad \text{(II. J. 5)}$$

(For notational convenience in this proof we have also assumed that at the start of each iteration, the jobs are renumbered so that $J_i$ is the $i^{th}$ job in F, $1 \leq i \leq k+N$, and the jobs $J_i$ with $i > k+N$ are not yet in F.) Obviously the inductive assumptions are satisfied at the start of the first iteration ($N = 0$), due to the initialization mentioned above.

Next consider the jobs $J_p$ with $D_p > 0$ (and of course $p > k$). At each step, F is a "partial" sequence satisfying (II. G. 1), and each iteration adds one or more jobs to the end of the partial sequence until it either becomes a complete sequence of all jobs in $\hat{J}$, or it is impossible to find a next job satisfying (II. G. 1).

Consider the block labelled L1 for some j. The test:

$$MARK_j \leq QSIZE_j \qquad \text{(II. J. 6)}$$

insures that the block labelled L2 will not be executed if all the $QSIZE_j$ elements in the $j^{th}$ column of Q have been checked and "approved" on previous iterations. The test:

$$D_{ij} \leq THRESH_j \qquad \text{(II. J. 7)}$$

is the $j^{th}$ component of condition (II. G. 1) (due to the value of $\overline{THRESH}$). Only if this test is true will the block labelled L3 be executed; if it is false the execution of block L1 for this value of j terminates, with the value of $MARK_j$ unchanged so that next time L1 is executed for this value of j, the same job $J_i$ will be tested again. Job $J_i$ is "approved" in block L3 by calling procedure APPROVE to decrement $DN_i$ by one in order to indicate that another one of the $DNUMB_i$ non-zero components of $\overline{D}_i$ has satisfied (II. J. 7). For each i and j this $DN_i$ is decremented at most once, because upon return from procedure APPROVE, the marker $MARK_j$ is immediately incremented by one, thereby pointing to the next element in column j of Q. Since $\overline{MARK}$ is never decremented or reset in this algorithm, and since the index i for job $J_i$ appears at most once in column j of Q, it is clear that this is the case. Control is then transferred back to the line labelled AGAIN in order to repeat this sequence of operations for the new value of $MARK_j$. For this value of j block L2 is executed repeatedly until the test (II. J. 6) fails or until the test (II. J. 7) fails, at which time the execution of block L1 for this j terminates. Note that as soon as the test (II. J. 7) fails for some job $J_i$ in column j, then it must also fail for all other jobs

following $J_i$ in column j, due to (II. J. 4). When execution of block L1 terminates for some j, $MARK_j$ retains the index of the next item in column j of Q.

Since each column j of Q is being tested in parallel, it is necessary to have global coordination of the separate approvals of a single job $J_i$. This is the purpose of making the procedure APPROVE a critical section. After this procedure has been called with parameter i a total of $DNUMB_i$ times, all $DNUMB_i$ components of $\overline{D}_i$ greater than zero are known to satisfy (II. J. 7), and hence:

$$\overline{D}_i \leq \overline{THRESH} = \overline{RFREE} + \sum_{p=1}^{k+N} \overline{A}_p, \qquad \text{(II. J. 8)}$$

which means that $J_i$ satisfies (II. G. 1) as the next job in $\overline{E}$. Since $DN_i$ was initialized to $DNUMB_i$, when it is decremented to zero in APPROVE, $J_i$ is added to the list $\overline{E}$ as the next job. On one execution of the statement REPEAT for all values of j, $\overline{DN}$ may be decremented to zero for several jobs, but the order in which these jobs are added to $\overline{E}$ is irrelevant since they all satisfy (II. J. 8). Therefore, if $J_i$ becomes the $k + 1^{st}$ job in $\overline{E}$ and $J_q$ becomes the $k + 2^{nd}$, $J_q$ will obviously satisfy:

$$\overline{D}_q \leq \overline{THRESH} + \overline{A}_i = \overline{RFREE} + \sum_{p=1}^{k+N} \overline{A}_p + \overline{A}_i \qquad \text{(II. J. 9)}$$

which is just (II. G. 1) for the $k + 2^{nd}$ job in $\overline{E}$ if $J_i$ is the $k + 1^{st}$.

When the statement labelled REPEAT has finished execution for all m values of j, the tests at L4 are made. If N has reached the value COUNT, then all jobs in $\hat{J}$ have satisfied (II. G. 1) and hence the complete sequence F has been

constructed. If N is not equal to COUNT, then there exists at least one job that has not satisfied (II. J. 7) in at least one component j. In this case, N is tested against OLDN, the value of N at the start of this iteration, to see if any job was added to the list $\overline{E}$ on this iteration. If not, then future repetition of REPEAT will be futile since the value of $\overline{THRESH}$ will not change and hence the results of test (II. J. 7) cannot change. Theorem 9 proves that if this happens, no sequence satisfying (II. G. 1) exists for system $\hat{S}$ at this time, and hence it is deadlocked. If however N > OLDN, then (N - OLDN) new jobs have been added to $\overline{E}$ on the iteration just completed. In this case block L5 is executed to add the $\overline{A}$ vectors of these newly added jobs to $\overline{THRESH}$, giving (II. J. 5) as the new value which is used at the start of the next iteration when control returns to REPEAT. Hence the conditions of F and $\overline{THRESH}$ assumed in the induction hypothesis hold as claimed, which proves the induction.

One final point should be made concerning the decrementing of $\overline{DN}$. It is both possible and probable that on any single iteration, $DN_i$ will be decremented one or more times, but will not reach zero, indicating that (II. J. 7) does not hold for all components of $\overline{D}_i$. However, for any components for which (II. J. 7) is true on this iteration, it will remain true on all subsequent iterations, since obviously each component of $\overline{THRESH}$ either remains constant or increases on each iteration, while the matrix D remains constant. Therefore there is never any need to "backup" and retest $D_{ij}$ once it has passed the test (II. J. 7). Q. E. D.

Theorem 9: If algorithm $L_1$ fails to generate a permutation sequence F of all jobs in $\hat{J}$ satisfying (II. G. 1), then no such sequence exists for system $\hat{S}$ at time t.

Proof:

Suppose $L_1$ fails to produce a sequence of all jobs in $\hat{J}$ that satisfies (II. G. 1), but that there does exist a sequence T of all jobs in $\hat{J}$ that does satisfy (II. G. 1) at time t. Let F be the partial sequence generated by $L_1$ up to the point it stopped, with the jobs renumbered so that $F = \{J_1, J_2, \ldots, J_k\}$, and $P = \{J_{k+1}, J_{k+2}, \ldots, J_n\}$. There must be at least one job in set P or the algorithm $L_1$ would not have failed, and obviously a job must be either in F or P, but not both. At the end of the iteration of $L_1$ that added $J_k$ to F the value of $\overline{THRESH}$ is given by (II. J. 3), and failure of any $DN_i$ to be reduced to zero on the next iteration implies that for each job $J_i$ in P (i.e., not yet added to F), at least one component j of $\overline{D}_i$ did not satisfy (II. J. 2) (this is true for all jobs in P due to the ordering of the columns of Q). Let $J_i$ be the first job in sequence T that is also in set P. This defines a set Q of all jobs that precede $J_i$ in T, and obviously Q is a subset of F, since by construction no job ahead of $J_i$ in T is in P. This gives:

$$\sum_{J_p \text{ in } Q} \overline{A}_p \leq \sum_{J_p \text{ in } F} \overline{A}_p = \sum_{p=1}^{k} \overline{A}_p$$

But since T satisfies (II. G. 1), by assumption, and $J_i$ is in T, we have:

$$\overline{D}_i \leq \overline{RFREE} + \sum_{J_p \text{ in } Q} \overline{A}_p \leq \overline{RFREE} + \sum_{p=1}^{k} \overline{A}_p = \overline{THRESH}$$

by (II. J. 3). But this means that job $J_i$ satisfies (II. J. 2) in all m components of $\overline{D}_i$, which contradicts the earlier assertion that (II. J. 2) does not hold for at least one component of $\overline{D}_p$ for every job $J_p$ in P, and $J_i$ is in P.

Therefore, a sequence T satisfying (II. G. 1) cannot exist at time t if algorithm $L_1$ fails. Q. E. D.

Theorem 10: Without advance information, algorithm $L_1$ determines uniquely whether or not system $\hat{S}$ is deadlock-free at time t.

Proof:

If $L_1$ succeeds in generating a complete permutation sequence of all jobs in $\hat{J}$, this sequence will satisfy (II. G. 1), by theorem 8, which implies that $\hat{S}$ is deadlock-free at time t, by theorem 5.

If $L_1$ fails to generate a complete permutation sequence of all jobs in $\hat{J}$, then no sequence of all jobs in $\hat{J}$ satisfying (II. G. 1) can exist for $\hat{S}$ at time t, by theorem 9, which implies that $\hat{S}$ is deadlocked, by theorem 3.

Q. E. D.

```
            begin
            parallel for j in [1,m] do
                  begin
                  THRESH_j := RAVAIL_j;
                  MARK_j := 1;
                  end;
            parallel for i in [1,n] do DN_i := DNUMB_i;
            N := OLDN := 0;
REPEAT:     parallel for j in [1,m) do
L1 :              begin
AGAIN:            if MARK_j ≤ QSIZE_j do
L2 :                    begin
                        i := Q_MARK_j,j;

                        if D_ij ≤ THRESH_j then
L3 :                          begin
                              APPROVE(i);
                              MARK_j +:= 1;
                              go to AGAIN;
                              end;
                        end;
                  end;
L4 :        if N = COUNT then <no deadlock> else
            if N = OLDN then <deadlock> else
L5 :              begin
                  for i from OLDN + 1 step 1 up to N do
                        parallel for j in [1,m] do THRESH_j +:= A_E_i,j;
                  OLDN := N;
                  go to REPEAT;
                  end;
            end;
```

Figure II. J. 1 Deadlock Detection Algorithm $L_1$

```
        critical section procedure APPROVE(i);
            begin
            DN_i -:= 1;
            if DN_i = 0 then
A1:             begin
                N +:= 1;
                E_N := i;
                end;
            end;
```

Figure II. J. 2 Procedure APPROVE used in Algorithm $L_1$.

## K. Cost Considerations and Algorithm $L_2$

### 1) The Cost of Algorithm $L_1$

In this subsection we will give an analysis of the costs associated with a single execution of the algorithm $L_1$. The next subsection describes the total cost of deadlock detection in system $\hat{S}$ when the overhead required by $L_1$ is taken into account, and the last subsection gives a new algorithm, $L_2$, designed to operate at a lower cost than $L_1$.

The initialization statements for algorithm $L_1$ (i.e., those statements preceding the label REPEAT in figure II.J.1) are executed only once, and will incur a fixed cost $C_0$, which is proportional to $n + m$. The statement labelled REPEAT will be executed a maximum of n times, with the maximum being achieved only in the case when a single job is added to list $\overline{E}$ on each iteration. The block labelled L1 will be entered m times on each iteration, once for each of the m values of j, giving an upper limit of $n \times m$ entries into the block. The test (II.J.6) on the line labelled AGAIN may be executed more than this due to the return of control to AGAIN from within block L3. It is clear however that over all iterations, the block labelled L3 can be entered at most n times for each of the m values of j, since each time this block is executed, $MARK_j$ is increased by one, and after n such increments its value will be $n + 1$ (since it is initialized to 1), which is larger than the maximum possible value of $QSIZE_j$ (see II.F.8), and hence test (II.J.6) must fail thereafter. If the cost of one execution of block L3 is $C_2$, then the maximum total cost of L3 to the entire algorithm is $m \times n \times C_2$.

Block L3 is executed only if both tests (II.J.7) and (II.J.6) are true, so both together can give a true result at most n times for a single value of j. If either test fails, that iteration for that value of j stops, so that the total

number of failures over all iterations of both tests together can be at most n for a single value of j (due to the fact that there can be at most n full iterations of the REPEAT statement). Therefore, if $C_a$ is the cost of test (II. J. 6), and $C_b$ is the cost of test (II. J. 7) plus the cost of assigning a value to i, the total cost of the entire block L1 will be at most: $m \times n \times (2 \times C_a + 2 \times C_b + C_2)$. Defining $C_1 = 2 \times C_a + 2 \times C_b$ as the cost of block L1 excluding block L3, this expression simplifies to $m \times n \times (C_1 + C_2)$.

To make $C_2$ a true constant, we must separate from the cost of block L3 the cost of executing the block labelled A1 in the procedure APPROVE, since this is executed only when $DN_i = 0$. This block is executed at most n times (once for each job $J_i$ with $DNUMB_i > 0$). In addition, the tests labelled L4 and the block labelled L5 can also be executed a maximum of n times, and this maximum occurs when only one job is added to $\overline{E}$ on each iteration. Grouping the costs of these three items into one figure $C_3$, a conservative maximum for the total cost of executing algorithm $L_1$ is: $C_0 + n \times (C_3 + m \times (C_1 + C_2))$.

In a practical situation, the average cost of this algorithm will be considerably less, due to the following considerations. The statement labelled REPEAT will never be executed a full n times, since only jobs $J_i$ with $DNUMB_i > 0$ are checked by this algorithm. If k jobs have $DNUMB_i = 0$, the maximum number of iterations is (n - k). Therefore the multiplier n in the above cost expression should be replaced by (n - k). Defining the factor $\mu = (n - k)/n$ $(0 \leq \mu < 1)$, the reduced total cost of $L_1$ becomes: $C_0 + \mu \times n \times (C_3 + m \times (C_1 + C_2))$.

A second cost reduction is achieved by noting that most jobs will not require elements from all m resource classes on each demand, but on the average will need elements from only $\ell$ different classes, $1 \leq \ell \leq m$. Therefore block L2 will be executed only $\ell$ times for each of the (n - k) jobs in $\overline{E}$, rather

than the m times described above, and hence the multiplier m of $C_2$ should be replaced by $\ell$. Defining the factor $\beta = \ell/m$ $(0 \leq \beta \leq 1)$ as the average fraction of the total number of resource classes needed in any single demand, the total cost of executing $L_1$ becomes: $C_0 + \mu \times (C_1 + m \times (C_1 + \beta \times C_2))$.

We should also note at this point that since there are an average of (n - k) jobs $J_i$ with $DNUMB_i > 0$, the total number of indices in the Q matrix will average about $\ell \times (n - k)$, and since there are m columns in Q, the average number of indices in a single column of Q will be $q = \ell \times (n - k)/m$. Defining the factor $\gamma = q/n$ as the average fraction of all jobs in a single column of Q $(0 \leq \gamma \leq 1)$, we get the relationship $\gamma = \beta \times \mu$, which will be useful later.

2) Overhead Costs for $L_1$

It is clear from the previous discussion that the cost of one execution of algorithm $L_1$ is proportional to n, rather than to $n^2$ as was true for Habermann's [12] algorithm, but it is also clear that algorithm $L_1$ requires more overhead bookkeeping than was necessary for his algorithm. It is important to show that these added costs do not erase the gain represented by $L_1$. In this section we will show that the additional costs due to the overhead are also proportional to n, and that they are added to the total cost of $L_1$, giving a total cost of deadlock detection in system $\hat{S}$ that is proportional to n.

The overhead for the deadlock detection algorithm consists of the bookkeeping necessary to maintain the vectors $\overline{RAVAIL}$ and $\overline{DNUMB}$, and the matrix Q. According to the primitive definitions given in section II.H all this extra bookkeeping is isolated in the two procedures PUTASLEEP and GETAWAKE. Referring to figure II.H.3, it is clear that the main blocks of these procedures (blocks labelled P1 and G1 respectively) are executed only once on each REQUEST, since once a job has been entered into list $\overline{E}$, further ASSIGNs will

not cause P1 to be executed, and G1 is executed only once at the time the job is to be removed from $\overline{E}$.

Within block P1 the only significant costs will be due to the m parallel executions of block P2, since the cost of ENQUEUE($J_k$, $\overline{E}$) is just the cost of the deadlock detection algorithm $L_1$ previously discussed. Within block P2, it must be noted that the ENQUEUE($J_k$, $\overline{Q}_j$) operation to enter a new item into column j of matrix Q may require up to n operations. Letting the cost of one of these operations be $C_4$, the cost of the ENQUEUE is $n \times C_4$. Letting $C_5$ be the cost of executing all other statements in block P2 once, the total cost of block P1 for the m values of j will be at most $m \times (n \times C_4 + C_5)$.

Due to the symmetry with PUTASLEEP, the costs for procedure GETAWAKE will be identical except for the additional cost of the DEQUEUE($J_k$,$\overline{E}$) when an item is to be removed from the list $\overline{E}$. Since there can be up to n jobs on this list, a single DEQUEUE may require up to n operations to remove an item and readjust the list, so that if the cost of one such operation is $C_6$, the total DEQUEUE cost is $n \times C_6$.

Therefore, the cost of executing blocks P1 and G1 once each on a single REQUEST is: $2 \times m \times (n \times C_4 + C_5) + n \times C_6$, plus the cost of the deadlock detection algorithm itself. Once again this is a conservative upper limit on the costs, since the same reduction factors discussed in the previous section are also applicable here. Therefore, we can replace the n multiplying $C_4$ with q, the average number of indices in a single column of Q; the n multiplying $C_6$ with (n - k), the average number of jobs in $\overline{E}$; and the m multiplying $C_4$ with $\ell$, the average number of different resource classes needed in each demand, to obtain as a conservative estimate of average cost:

$2 \times m \times (n \times \beta \times \gamma \times C_4 + C_5) + n \times \mu \times C_6$, plus the cost of algorithm $L_1$.

This average can be reduced still further by noting that not every REQUEST requires the execution of blocks P1 and G1. If the requested number and type of resource elements are free at the time of the REQUEST, neither P1 or G1 is executed. Letting $\alpha$ be the ratio of the number of REQUESTs causing P1 and G1 to be executed to the total number of REQUESTs, the total cost of a single REQUEST will be reduced by the factor $\alpha$. (Clearly $0 \leq \alpha \leq 1.$) This gives as the average cost per REQUEST for deadlock detection with algorithm $L_1$:

$$\alpha \times (2 \times m \times (n \times \beta \times \gamma \times C_4 + C_5) + n \times \mu \times C_6 +$$

$$C_0 + \mu \times n \times (C_3 + m \times (C_1 + \beta \times C_2))).$$

Regrouping the terms, and using the relationship $\gamma = \beta \times \mu$, we get:

$$\alpha \times (C_0 + 2 \times m \times C_5 + n \times \mu \times (C_6 + C_3 + m \times (C_1 + \beta \times C_2 + 2 \times \beta^2 \times C_4))).$$

3) Algorithm $L_2$

We can reduce the costs of deadlock detection even further by taking advantage of a special property of sequences satisfying (II. G. 1). If system $\hat{S}$ is deadlock-free at time t and job $J_k$ makes a REQUEST that cannot be immediately satisfied, system $\hat{S}$ remains deadlock-free at time t' immediately after the bookkeeping in procedure PUTASLEEP if and only if there exists a partial permutation sequence F' that satisfies (II. G. 1) and ends with job $J_k$. It is proved in theorem 11 that existence of such a partial permutation sequence, coupled with the known fact that system $\hat{S}$ was deadlock-free before $J_k$ made its REQUEST, uniquely determines whether or not system $\hat{S}$ remains deadlock-free at time t'. An algorithm to produce such a partial sequence will reduce the number of iterations of the statement labelled REPEAT by an average amount $\rho$, $0 < \rho \leq 1$.

Theorem 11: Without advance information, a system $\hat{S}$ that is deadlock-free at time t remains deadlock-free after an unsatisfiable REQUEST by job $J_k$ at time t if and only if there exists a partial permutation sequence of jobs that satisfies (II. G. 1) and ends with job $J_k$.

Proof:

If a partial sequence ending in $J_k$ does not exist, then a complete permutation sequence of all jobs in $\hat{J}$ also cannot exist, which by theorem 3 means that $\hat{S}$ is deadlocked at time t.

Suppose there does exist a partial sequence F' that satisfies (II. G. 1) and ends in $J_k$. We show how to make this into a complete sequence of all jobs in $\hat{J}$, which by theorem 5 makes $\hat{S}$ deadlock-free. Let P be the set of jobs of $\hat{J}$ that are not in F', with all jobs renumbered so that $F' = \{J_1, J_2, \ldots, J_k\}$, and $P = \{J_{k+1}, J_{k+2}, \ldots, J_n\}$. The complete sequence is constructed by repeatedly removing a job from P and adding it to the end of F' so that (II. G. 1) is satisfied at each step.

Between time t when job $J_k$ invoked the REQUEST and time t' immediately after the bookkeeping in PUTASLEEP has been performed, the only change which has occurred in data base $B_1$ is that $\overline{D}_k(t') > 0$, whereas $\overline{D}_k(t) = 0$. Let F be a sequence satisfying (II. G. 1) for system $\hat{S}$ at time t (by theorem 1 there must be at least one such sequence F if $\hat{S}$ is deadlock-free at time t), and let $J_i$ be the first job in F that also belongs to P. This defines a set Q of all jobs that precede $J_i$ in F. Jobs in Q cannot be in P, so they must belong to F', by the definition of P and the selection of $J_i$. Therefore Q is a subset of F', and :

$$\sum_{J_p \text{ in } Q} \overline{A}_p(t') \leq \sum_{J_p \text{ in } F'} \overline{A}_p(t') = \sum_{p=1}^{k} \overline{A}_p(t')$$

Since $J_i$ satisfied (II. G. 1) in its position in F at time t, and only $\overline{D}_k$ changed between time t and time t', we have:

$$\overline{D}_i(t') \leq \overline{RFREE}(t') + \sum_{J_p \text{ in } Q} \overline{A}_p(t') \leq \overline{RFREE}(t') + \sum_{p=1}^{k} \overline{A}_p(t')$$

Thus $J_i$ satisfies (II. G. 1) as the $k + 1^{st}$ job to be added to F' to form a new partial sequence F'' of k + 1 jobs satisfying (II. G. 1). To find the next job to add to F'', simply remove $J_i$ from P to give P', and repeat the above proof for F'' and P'. This should be continued until all jobs originally in P have been added to the sequence to give a complete permutation sequence of all jobs in $\hat{J}$ that satisfies (II. G. 1) by construction, and hence $\hat{S}$ is deadlock-free by theorem 5. Q. E. D.

The new algorithm is almost identical to the previous one, with most changes occurring in the procedure APPROVE. The new version of APPROVE is given in figure II. K. 1. The basic changes are as follows:

1) The test labelled A3 in figure II. K. 1 is added to check whether or not the job for which DN is decremented to zero is $J_k$, the job being enqueued on the $\overline{E}$ list in the call ENQUEUE($J_k$, $\overline{E}$). If it is, the algorithm should terminate immediately with an indication of no deadlock.

2) Since the new algorithm would generate only a partial sequence in $\overline{E}$, ending with job $J_k$, it becomes necessary to add a mechanism to construct the full schedule in $\overline{E}$, since $\overline{E}$ is defined in item 10 to be a list of all jobs $J_i$ with $DNUMB_i = 0$. It is shown in the proof of theorem 11 that this can be done by simply appending the remaining jobs onto

$\overline{E}$ in the order in which they appeared in the previous schedule. This requires the following:

a) Include in the temporary storage used by $L_2$ an n-component vector called $\overline{OLDE}$, used to hold a copy of the previous state of vector $\overline{E}$; and an n-component bit vector called $\overline{Y}$, used to mark each job as it is entered in the new list $\overline{E}$.

b) Add to the initialization phase of $L_2$ the statement:

```
parallel for i in [1, n] do
    begin
    OLDE_i := E_i;
    Y_i := 1;
    end;
```

This is the only modification necessary to algorithm $L_1$ as shown in figure II.J.1 to give algorithm $L_2$. All other changes occur in procedure APPROVE.

c) Include in APPROVE the statement labelled A2 in figure II.K.1, in order to change the value of $Y_i$ from one to zero whenever $DN_i$ is decremented to zero.

d) Add the block labelled A4 to APPROVE in order to place all elements of $\overline{OLDE}$ whose corresponding $\overline{Y}$ value is not zero on to $\overline{E}$ in the same sequence as they appear in $\overline{OLDE}$. This is done only once, at the time $DN_k$ goes to zero and the partial sequence ending in $J_k$ is complete in $\overline{E}$.

Algorithm $L_2$ is used identically to $L_1$ and the expected cost reduction factor $\rho$ can be applied to the entire cost of $L_1$ derived in subsection 1, excluding the initialization cost $C_0$ which actually increases slightly due to the required initialization of $\overline{OLDE}$ and $\overline{Y}$. The cost $C_3$ will also have to be increased slightly to reflect the additional statement A2 and the test for i = k that are now required for each execution of APPROVE. Finally a new cost must be added for the new block A4 in APPROVE. This block is executed only once, when the job $J_k$ is added to $\overline{E}$, so that its contribution to the total cost will be proportional to COUNT, the number of times that the statement labelled A5 is executed. Since COUNT has an average value of $\mu \times n$, if we let $C_7$ be the cost of one execution of the statement labelled A5 (including A6), the total cost of block A4 will be $\mu \times n \times C_7$. This gives as the total cost per REQUEST for deadlock detection using algorithm $L_2$:

$$\alpha \times (C_0 + 2 \times m \times C_5 + n \times \mu \times (C_6 + C_7 + \rho \times C_3 +$$

$$m \times (\rho \times C_1 + \rho \times \beta \times C_2 + 2 \times \beta^2 \times C_4)))$$

The costs and cost reduction factors are summarized in figure II. K. 2, along with the average cost per REQUEST of deadlock detection with algorithms $L_1$ and $L_2$. The costs $C_0$ - $C_7$ depend on the particular hardware being used to perform the algorithms, and are therefore constant only for a given processor. The factors written as Greek letters are all fractions between zero and one that reflect the average use of the system. The more loaded the system (i.e., the more jobs there are in relation to total resources) the closer the factors $\alpha$, $\mu$, $\gamma$ and probably also $\rho$ will be to one.

It should be pointed out that the average cost per REQUEST does not take into account the high degree of parallelism in these algorithms. The cost is represented in terms of the total number of operations, which is proportional to the total number of time units to complete the operations when all the parallelism is taken out; that is, when there is only a single processor that must execute everything in a serial manner. If m processors were available (one for each resource class), the factor of m could be eliminated from the cost expressions in figure II. K. 2, although an additional small cost with a factor of $m \times n \times \mu \times \beta$ would have to be added to account for the disruption to the parallelism caused by the critical section procedure APPROVE, which cannot be executed simultaneously by several processors.

Finally we should mention the amount of additional storage needed to detect deadlock, since it was by increasing the amount of stored information about the system status that we were able to decrease the time needed by the deadlock detection algorithms. As mentioned in section II. F, the permanent data structures in data base $B_1$ that exist solely for use in deadlock detection are the n by m matrix Q, the m-component vectors $\overline{QSIZE}$ and $\overline{RAVAIL}$, and the n-component vector $\overline{DNUMB}$. In addition algorithms $L_1$ and $L_2$ require for working storage the m-component vectors $\overline{MARK}$ and $\overline{THRESH}$, the n-component vector $\overline{DN}$, and for $L_2$ only, the n-component vectors $\overline{OLDE}$ and $\overline{Y}$. Thus the total additional storage for deadlock detection with algorithm $L_1$ is $2 \times n + n \times m + 4 \times m$ cells, and for algorithm $L_2$ it is $4 \times n + n \times m + 4 \times m$ cells. This represents the extra storage that is "traded off" for the extra speed of algorithms $L_1$ and $L_2$.

```
critical section procedure APPROVE(i);
    begin
    DN_i -:= 1;
    if DN_i = 0 then
A1:     begin
        N +:= 1;
        E_N := i;
A2:     Y_i := 0;
A3:     if i = k then
A4:         begin
            for i from 1 step 1 up to COUNT do
A5:             if Y_OLDE_i > 0 then
A6:                 begin
                    N +:= 1;
                    E_N := OLDE_i;
                    end;
            <stop, no deadlock>
            end;
        end;
    end;
```

Figure II. K. 1 Procedure APPROVE used in Algorithm $L_2$.

$C_0$--cost of all statements in the initialization phase of the deadlock detection algorithms.

$C_1$--cost of one execution of block L1, excluding block L3.

$C_2$--cost of one execution of block L3 including the cost of procedure APPROVE except for block A1.

$C_3$--cost of one execution of block A1 plus the cost of the tests L4 plus block L5.

$C_4$--cost of one operation used by ENQUEUE and DEQUEUE operating on columns of matrix Q.

$C_5$--cost of one execution of block P2 or G2 excluding the cost of ENQUEUE and DEQUEUE on list $\overline{E}$.

$C_6$--cost of one operation in a DEQUEUE on list $\overline{E}$.

$C_7$--cost of one execution of statement A5.

---

$\alpha$--average fraction of the total number of REQUESTs that cannot be satisfied at the time of issue.

$\beta$--average fraction of the total number of resource classes needed in any one REQUEST.

$\gamma$--$\beta \times \mu$, average fraction of all jobs in any one column of Q.

$\rho$--average ratio of the number of iterations of statement REPEAT by $L_2$ to the number of iterations required by $L_1$.

$\mu$--average fraction of all jobs in list $\overline{E}$.

---

Average Cost per REQUEST of Deadlock Detection with Algorightm $L_1$:

$$\alpha \times (C_0 + 2 \times m \times C_5 + n \times \mu \times (C_6 + C_3 + m \times (C_1 + \beta \times C_2 + 2 \times \beta^2 \times C_4)))$$

Average Cost per REQUEST of Deadlock Detection with Algorithm $L_2$:

$$\alpha \times (C_0 + 2 \times m \times C_5 + n \times \mu \times (C_6 + C_7 + \rho \times C_3 + m \times (\rho \times C_1 + \rho \times \beta \times C_2 + 2 \times \beta^2 \times C_4))).$$

---

Figure II.K.2 Cost Summary Table for Deadlock Detection in System $\hat{S}$.

## Chapter III. Deadlock Detection with Simultaneous Resource Sharing

### A. Introduction

Assumption 5 in Section II.E states that at any instant of time a resource element can be controlled by at most one job. With such an assumption resource elements are considered to be "serially reusable" by different jobs, but there is no possibility of two or more jobs simultaneously sharing access to a common resource. This has been true of all previous research on deadlock detection and prevention except for Murphy's [23]. He considers the special case where each resource class contains exactly one element, but this element can be controlled by jobs in one of two modes: (1) exclusive control mode, where there is exactly one job in control and no other job can obtain access to the element until the controlling job releases it; (2) shared control mode, where two or more jobs are permitted to have simultaneous access to a single element. Murphy also limits his discussion to a system in which resource requests must be serviced in a first-come, first-serve (FIFO) manner. As has been discussed by Holt [20], this can cause unnecessary deadlocks, since a sequence of allocations that does not lead to a deadlock situation may exist, but the FIFO queuing discipline prevents the scheduler from allocating resources in the deadlock-free manner. Clearly deadlocks can arise without requiring that the scheduler follow a specific queuing discipline, and in all our discussions here we will assume there is no such imposed ordering.

It has been claimed, by Denning [5,6] in particular, that simultaneous sharing of common resource elements can be extremely advantageous to the efficiency of a computer system. However, since this concept has been ignored in most previous research on deadlock, it may be useful at this point

to consider some cases where the need for such a facility may arise. The most obvious case will be the one in which a resource class contains a single element, such as a named data file in a file system, that can be either read or written. If a job only wishes to read the file, it can share access with any other job also wishing to read it; but if a job is going to write into the file, no other job can be permitted to either read or write into that file simultaneously, since the results would in general not be well defined [37]. Therefore, when a job requests access to the file, it must also specify the mode of control desired, so that the scheduler can grant the allocation with the proper access mode.

Simultaneous sharing of resource elements in the general case of a resource class containing more than one element has less obvious but still quite practical applications in large multi-process systems. Depending on how "resources" are defined in a system, these classes will usually be "infinite sources" (system input units), "infinite sinks" (system output units), or "infinite buffers" (system scratch units). For example, a system may be designed so that each disk drive plus its managing software is considered to be a separate resource element in the class of disk resources. A job may then require several such resource elements, but due to the internal logic of the disk manager, several jobs may share access to the same element simultaneously with no difficulty (the queuing and identification of data items being sent to and from the disk is handled internally by the manager). On the other hand, if the user wishes to use his own private disk pack, his job must have exclusive control of one of these resource elements in order to guarantee that no other job will attempt to read or write on his private files. In both cases, the job simply REQUESTs any available disk resource, and is not

particular about which of the existing elements in the class it is assigned. (Obviously once the assignment is made the job must be able to uniquely identify the element it controls, in this example so that the private disk pack can be mounted on the proper drive.) It is of course possible that a single job may require several disk resources in both modes of control. It is also obvious that the same resource element may at one time be allocated in exclusive control mode, and at another time in shared control mode.

In order to permit resource elements to be shared simultaneously, we must first replace assumption 5 with assumption 5' as was described in section II.E. The notation $\hat{S}'$ will be used to indicate systems based on assumption 5' rather than 5 (which was indicated by $\hat{S}$). The new state diagram for resource elements in system $\hat{S}'$ is given in figure III.A.1. The state diagram for jobs in $\hat{S}'$ remains the same as in $\hat{S}$ (see figure II.B.1). Changes will also be required to the data base B, to the primitives defining scheduler S, and finally to the deadlock detection algorithms in order to reflect the extra work necessary for the two modes of resource control. This is discussed next.

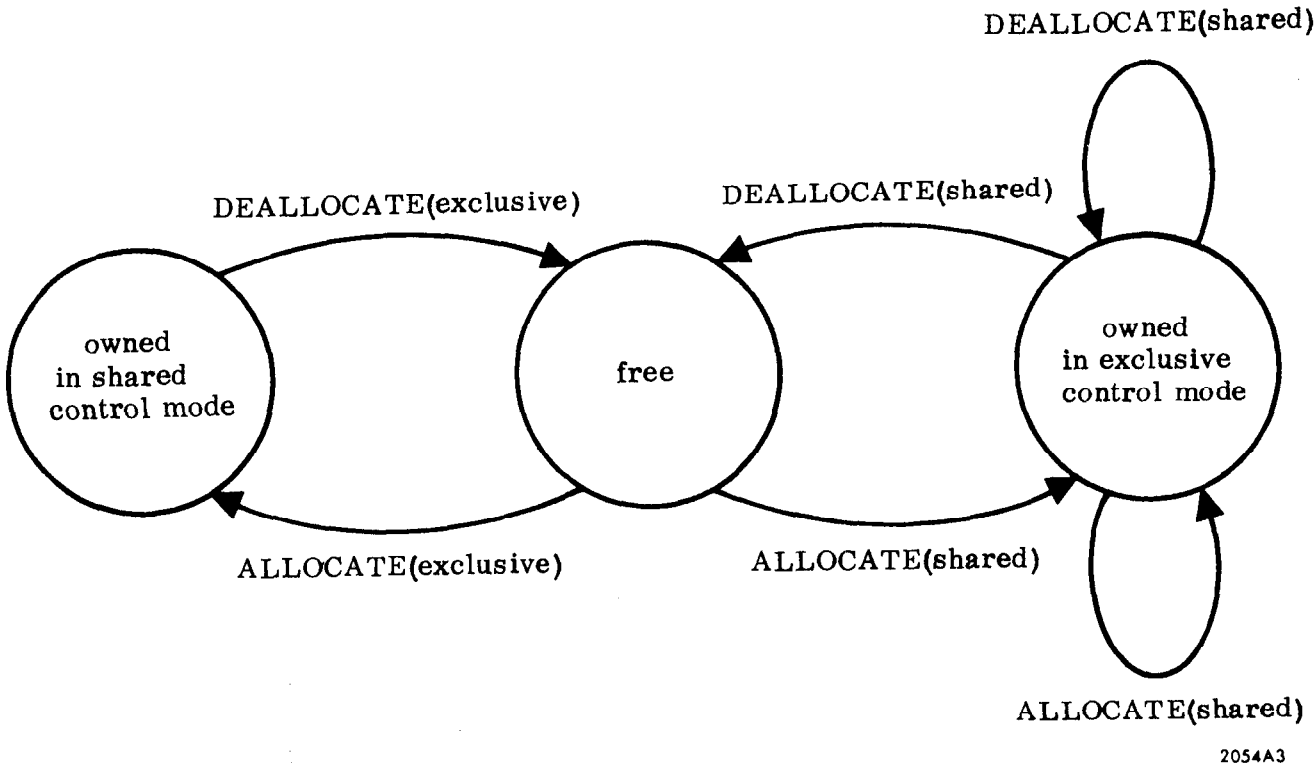


Figure III.A.1 State Diagram for Resource Elements in System $\hat{S}$.

B. Data Base $B_2$.

The data base $B_2$ which is part of system $\hat{S}'$ is shown schematically in figure III.B.1. Because a job can request and be assigned resource elements in one of two modes, the first obvious difference between $B_1$ and $B_2$ is the replacement of the assignment matrix A by two matrices: AE, for keeping track of elements controlled in exclusive mode; and AS, for keeping track of elements controlled in shared mode. Similarly, the demand matrix D will be replaced by two matrices, DE and DS, and the request queue matrix Q will be replaced by the matrices QE and QS for use by the new deadlock detection algorithm. For convenience, we will now use the notation that matrix A represents the n by 2m matrix formed from AE and AS — the first m columns of A are AE, the second m are AS. Similarily, D represents DE and DS, and Q represents QE and QS. Therefore, an expression such as $\overline{D}_i \geq 0$ means both $\overline{DE}_i \geq 0$ and $\overline{DS}_i \geq 0$. In addition, the vector $\overline{QSIZE}$ now represents two m-component vectors, $\overline{QESIZE}$ and $\overline{QSSIZE}$, which relate to QE and QS respectively in such a manner that the original relationship of $\overline{QSIZE}$ to Q (see item 8) is maintained.

Continued use of the notation A, D, Q, and $\overline{QSIZE}$ reflects the fact that, as far as much of the bookkeeping for these matrices is concerned, the effect of allowing two modes of control is simply to split each of the original m resource classes into two subclasses, the shared class and the exclusive class, so that the original bookkeeping functions pertaining to owned resources must now be applied to 2m possible subclasses of owned resource elements. However, since these two subclasses of each class are not completely independent, as are two different classes, additional bookkeeping will be required to handle the assignment of each existing element to one of the two possible

subclasses when it becomes owned. In addition, the new deadlock detection algorithm requires that the ordering of items in the columns of Q be defined somewhat differently than before (see definition 7).

The vector $\overline{\text{DNUMB}}$ must be defined so that it will still reflect the total number of resource classes needed in a demand, regardless of the mode of control being demanded. Hence in system $\hat{S}'$, the $i^{th}$ component of $\overline{\text{DNUMB}}$ is defined as the number of positive elements in the $i^{th}$ row of (DE+DS), which is <u>not</u> $\bar{D}_i$. This reflects the fact that demands to the same resource class, although in different modes, are not independent, since both must be satisfied simultaneously from a single set of actual resource elements.

We must also introduce three new data structures that will be necessary to maintain information on resource sharing.

<u>Item 11</u>: $\overline{\text{RSHARE}}$--an m-component vector whose $j^{th}$ component is the number of resource elements of type j currently <u>owned</u> in shared control mode.

We assume that an allocation policy will try to minimize the number of different resource elements of the same type that are shared simultaneously, so that when a request for a shared resource is encountered, elements that are already owned in shared control mode by other jobs will be assigned first, and free resources will be used only after every shared resource element of the proper class has been assigned to the job. However, in spite of this policy it is still possible for two or more jobs to have shared control of a disjoint set of resource elements of the same type. For example, if job $J_1$ has shared control of elements a and b, and job $J_2$ has control of a in shared mode, then if $J_1$ releases control of a but not b, we are left with the situation that a is controlled in shared mode by $J_2$ only, b is controlled in shared mode

by $J_1$ only, and $a \neq b$. This gives us upper and lower bounds on $\overline{\text{RSHARE}}$, the total number of different elements shared at any one time:

$$\sum_{i=1}^{n} \overline{AS}_i(t) \geq \overline{\text{RSHARE}}(t) \geq \max_i \overline{AS}_i(t) \qquad \text{(III.B.1)}$$

where we use the notation that the maximum of a set of vectors is the vector of the maximum of the corresponding elements. The new definition of $\overline{\text{RSHARE}}$ requires a redefinition of $\overline{\text{RFREE}}$ and $\overline{\text{RAVAIL}}$ as follows:

$$\overline{\text{RFREE}} = \overline{\text{RMAX}} - \sum_{i=1}^{n} AE_i - \overline{\text{RSHARE}} \qquad \text{(III.B.2)}$$

$$\begin{aligned} \overline{\text{RAVAIL}} &= \overline{\text{RFREE}} + \sum_{\text{DNUMB}_i=0} \overline{AE}_i + \overline{\text{RSHARE}} \\ &= \overline{\text{RMAX}} - \sum_{\text{DNUMB}_i>0} \overline{AE}_i \end{aligned} \qquad \text{(III.B.3)}$$

Item 12: SC--an s by m matrix, where $s = \max_{1 \leq j \leq m} \text{RMAX}_j$. (i.e., s is the maximum number of resource elements in the largest resource class.) The value of element $SC_{ij}$ is the number of jobs $J_k$ with $\text{DNUMB}_k > 0$ that have shared control of resource element i in class $R_j$.

The SC matrix will be used by the deadlock detection algorithm, and enables that algorithm to be linear in n.

Item 13: $\overline{\text{SCNUMB}}$--an m-component vector whose $j^{th}$ component is the number of positive elements in the $j^{th}$ column of SC.

$\text{SCNUMB}_j$ will be the total number of resource elements of type j controlled in shared mode by at least one job $J_k$ with $\text{DNUMB}_k > 0$. Since this is clearly

only a subset of all jobs in $\hat{J}$, we have:

$$\overline{\text{RSHARE}}(t) \geq \overline{\text{SCNUMB}}(t) \geq \max_{\text{DNUMB}_i>0} \overline{\text{AS}}_i \qquad \text{(III.B.4)}$$

Looking ahead to the bookkeeping that will be necessary in the primitive definitions to maintain SC, $\overline{\text{SCNUMB}}$, and $\overline{\text{RAVAIL}}$, it should be clear that these will change value only when $\text{DNUMB}_i$ for some job $J_i$ changes from zero to non-zero or vice versa. Hence the bookkeeping for these data structures will be contained entirely within the procedures PUTASLEEP and GETAWAKE, since these are called only when a job $J_i$ is put asleep due to insufficient resources to satisfy a REQUEST ($\text{DNUMB}_i$ goes from zero to non-zero), and when $J_i$ is gotten awake at some later time when sufficient resources have become available to satisfy its demand ($\text{DNUMB}_i$ goes from non-zero to zero). This is exactly what would be expected, since these data structures exist in the data base solely for utilization by the deadlock detection algorithm (and deadlock can only occur when a job is put asleep). However, the bookkeeping for $\overline{\text{RSHARE}}$, as for $\overline{\text{RFREE}}$, is essential to the scheduler for $\hat{S}'$, whether or not deadlock detection is to be included. Its maintenance must necessarily be incorporated into the primitive definitions themselves, rather than being isolated in auxiliary functions. See section III.D.

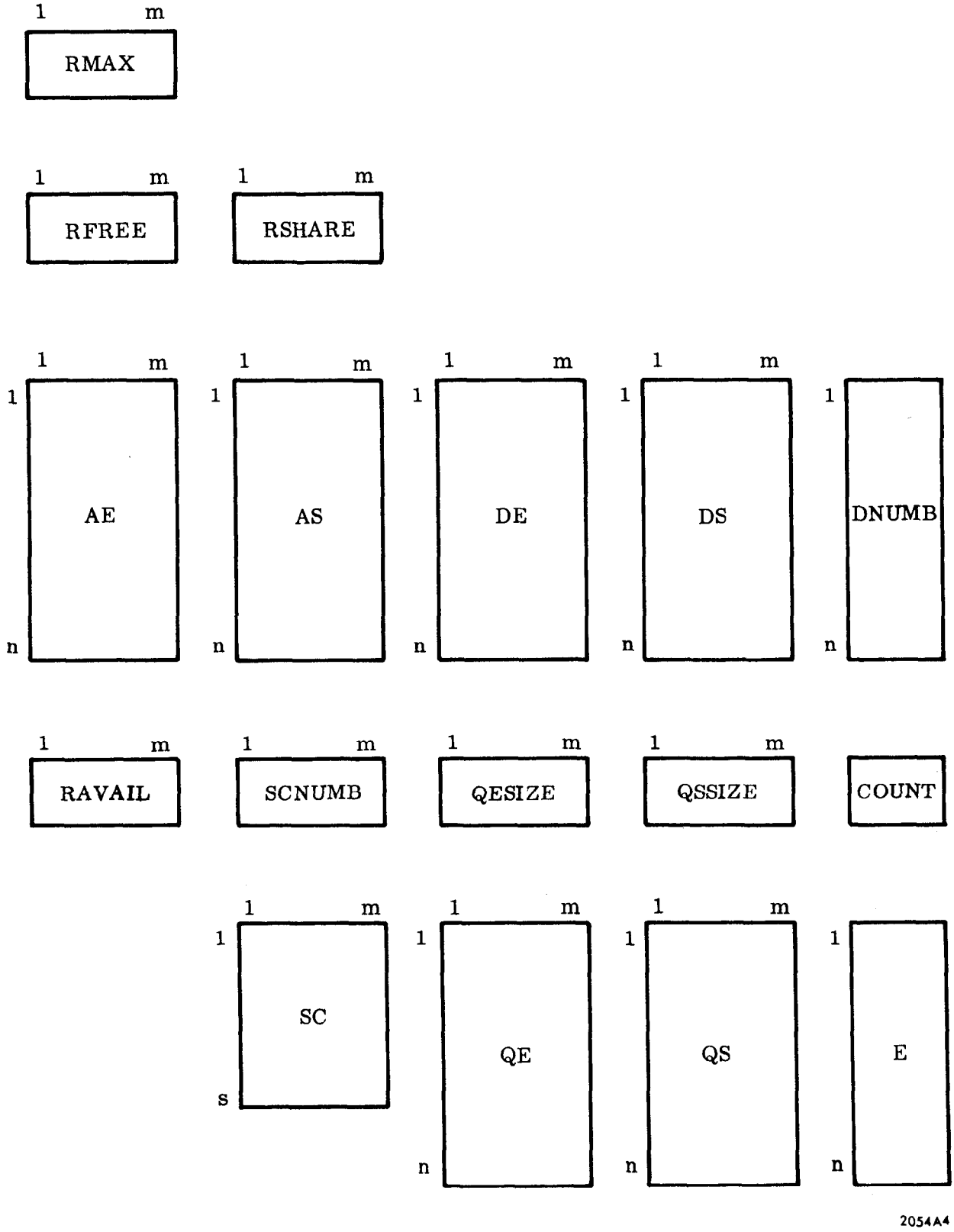


Figure III. B. 1 Scheduler Data Base $B_2$ for System $\hat{S}'$.

## C. Formal Properties of $\hat{S}'$

At any time t, the number of resource elements available for assignment to job $J_i$ in exclusive control mode is $\overline{RFREE}(t)$, the resources not yet assigned to any job, and the number available for assignment to $J_i$ in shared control mode is $(\overline{RSHARE}(t) - \overline{AS}_i(t))$, the number shared by other jobs but not $J_i$, plus $(\overline{RFREE}(t) - \overline{DE}_i(t))$, the number of free elements that remain after the exclusive assignment to $J_i$ is satisfied. Hence the conditions which must be satisfied in order to be able to satisfy a demand $\overline{D}_i(t)$ by job $J_i$ at time t are:

$$\overline{DE}_i(t) \leq \overline{RFREE}(t)$$

$$\overline{DS}_i(t) + \overline{AS}_i(t) \leq \overline{RFREE}(t) + \overline{RSHARE}(t) - \overline{DE}_i(t) \qquad \text{(III.C.1)}$$

Although definitions 1-5 and theorem 1 of section II.G remain valid for system $\hat{S}'$, the conditions that must be satisfied by a finishing sequence of $\hat{S}'$ must be redefined in order to reflect the new conditions (III.C.1) that must be satisfied before an assignment can be made. The new conditions are given in the next theorem, which is the analogy of theorem 2. Following this theorem, the new conditions are used to prove theorems 13, 14, and 15, which are analogous to theorems 3, 4, and 5 respectively.

Unless stated otherwise, we assume in all that follows that the jobs in $\hat{J}$ are numbered so that $J_k$ is the $k^{th}$ job in the permutation sequence F.

Theorem 12: If F is a finishing sequence for system $\hat{S}'$ at time t, then the following conditions are satisfied by jobs in F:

$$\overline{DE}_i(t) \leq \overline{Z}_i(t) - \overline{RS}_i(t)$$

$$\overline{DS}_i(t) + \overline{AS}_i(t) \leq \overline{Z}_i(t) - \overline{DE}_i(t) \qquad i = 1, 2, \ldots, n \qquad \text{(III.C.2)}$$

where we define the terms $\overline{Z}_i$ and $\overline{RS}_i$ as follows:

$$\overline{Z}_1(t) = \overline{RFREE}(t) + \overline{RSHARE}(t)$$

$$\overline{Z}_i(t) = \overline{RFREE}(t) + \sum_{k=1}^{i-1} \overline{AE}_k(t) + \overline{RSHARE}(t) = \overline{Z}_{i-1}(t) + \overline{AE}_{i-1}(t) \qquad \text{(III.C.3)}$$

for $i = 2, 3, \ldots, n$

$\overline{RS}_i$ is an m-component vector such that $RS_{ij}$ is the number of resource elements of type j which are controlled in shared mode by at least one job $J_k$ with $k \geq i$ (i.e., $J_k = J_i$ or $J_k$ follows $J_i$ in F).

From this definition we have the following relationships:

$$\sum_{k=i}^{n} \overline{AS}_k \geq RS_i \geq \max_{k \geq i} \overline{AS}_k$$

$$\sum_{k=1}^{n} \overline{AS}_k \geq \overline{RSHARE} = \overline{RS}_1 \geq \max_{k} \overline{AS}_k \qquad \text{(III.C.4)}$$

Neither Z nor RS is represented in data base $B_2$ because they will be computed when needed in algorithm $L_3$. They are both clearly a function of the position of a job in a particular sequence F, and will be written as $\overline{Z}_i(F,t)$ and $\overline{RS}_i(F,t)$ whenever necessary for clarity.

Proof (by induction):

Let F be a finishing sequence for $\hat{S}'$ at time t and assume that $J_1$, the first job in F, does not satisfy (III.C.2), so that $DNUMB_1 > 0$ and for at least one value of j, either:

$$DE_{1j}(t) > Z_{1j}(t) - RS_{1j}(t) = RFREE_j(t) + RSHARE_j(t) - RSHARE_j(t) = RFREE_j(t)$$

or

$$DS_{1j}(t) + AS_{1j}(t) > Z_{1j}(t) - DE_{1j}(t) = RFREE_j(t) + RSHARE_j(t) - DE_{1j}(t)$$

or both. Hence (III.C.1) does not hold for $J_1$ at time t, which means that an assignment cannot be made to $J_1$ at time t. This remains true until $\overline{RFREE}$ or $\overline{RSHARE}$ or both increase, which can only happen by a RELEASE by some other job $J_k$, $k \neq 1$. But $J_k$ must have $DNUMB_k = 0$ (i.e., be active) at time $t_k$, $t \leq t_k < t_1$, in order to be able to invoke a RELEASE at time t', $t_k \leq t'$. Therefore, at time $t_k$ we have $DNUMB_1(t_k) > 0$ and $DNUMB_k(t_k) = 0$, so that $J_k$ must precede $J_1$ in F, by definition 5. Contradiction of our assumption that $J_1$ is the first job in F. Hence (III.C.2) must hold for $J_1$.

Now assume that (III.C.2) holds for $J_1, J_2, \ldots, J_{i-1}$, but not $J_i$, so that $DNUMB_i > 0$. Between time t and $t_{i-1}$, only jobs $J_k$ with $k < i$ will achieve $\overline{D}_k = 0$, by definition 5. If all these jobs had released control of all their resources by time t', $t_{i-1} \leq t'$, we would have the following situation (since no other job $J_k$ with $k \geq i$ has $DNUMB_k = 0$ (by definition 5) and hence $\overline{D}_k$ and $\overline{A}_k$ remain unchanged between time t and t').

The maximum number of shared resources still assigned at time t' will be those controlled in shared mode by jobs $J_k$ with $k \geq i$. This is simply $\overline{RS}_i(t')$, which has not changed since time t, so that:

$$\overline{RS}_i(t) = \overline{RS}_i(t') \geq RSHARE(t') .$$

The maximum number of free resources at time t' are all those free at time t (i.e., RFREE(t)), plus those assigned in exclusive control mode to jobs $J_k$, $k \leq i-1$ (i.e., $\sum_{k=1}^{i-1} \overline{AE}_k(t)$), plus those shared only

by jobs $J_k$ with $k \leq i-1$ (i.e., $\overline{RSHARE}(t) - \overline{RS}_i(t)$). This gives us:

$$\overline{RFREE}(t') \leq \overline{RFREE}(t) + \sum_{k=1}^{i-1} \overline{AE}_k + \overline{RSHARE}(t) - \overline{RS}_i(t)$$

$$= \overline{Z}_i(t) - \overline{RS}_i(t)$$

If $J_i$ is to be the next job in F, we must reduce $\overline{D}_i$ to zero at some time $t_i \geq t'$. To make an assignment to $J_i$ at time t', condition (III.C.1) must hold, which would give:

$$\overline{DE}_i(t) = \overline{DE}_i(t') \leq \overline{RFREE}(t') \leq \overline{Z}_i(t) - \overline{RS}_i(t)$$

and

$$\overline{DS}_i(t) + \overline{AS}_i(t) = \overline{DS}_i(t') + \overline{AS}_i(t') \leq \overline{RFREE}(t') + \overline{RSHARE}(t') - \overline{DE}_i(t')$$

$$\leq \overline{RS}_i(t) + \overline{Z}_i(t) - \overline{RS}_i(t) - \overline{DE}_i(t)$$

$$= \overline{Z}_i(t) - \overline{DE}_i(t)$$

But these are exactly conditions (III.C.2) which we assumed did not hold for $J_i$. Therefore an assignment cannot be made to $J_i$ at any time $t_i \geq t'$ until either $\overline{RFREE}$ or $\overline{RSHARE}$ or both increase, which can only happen if some other job $J_k$ invokes a RELEASE, and $k > i$ (since all jobs $J_k$ with $k < i$ are already assumed to have released their resources by time t'). This job $J_k$ would have to achieve $\overline{D}_k(t_k) = 0$ at some time $t_k$ in order to RELEASE any resources at time $t'_k$, and $t_k \leq t'_k$. Therefore at time $t_k$ we have $DNUMB_i(t_k) > 0$, and $DNUMB_k(t_k) = 0$, which implies that $J_k$ must precede $J_i$ in F, by definition 5.

Contradiction of our assumption that $J_i$ is the next job in F. Hence (III.C.2) must also hold for $J_i$, which proves the induction.

Q.E.D.

Theorem 13: If at time t there does not exist a permutation sequence F of all jobs in $\hat{J}$ that satisfies conditions (III.C.2), then system $\hat{S}'$ is deadlocked at time t.

Proof:

Obvious, since theorem 12 proves that conditions (III.C.2) are necessary to any finishing sequence of $\hat{S}'$, and if no finishing sequence exists then $\hat{S}'$ is deadlocked at time t, by theorem 1. Q.E.D.

Theorem 14: Without advance information, any permutation sequence F of jobs in $\hat{J}$ that satisfies (III.C.2) at time t is a finishing sequence for system $\hat{S}'$ under the assumption that no new resource REQUESTs will be made at any future time $t' > t$.

Proof (by induction):

$J_1$ is the first job in F, so that (III.C.2) implies that (III.C.1) holds at time t for $J_1$. Thus if $DNUMB_1(t) > 0$, it is possible to make an assignment to $J_1$ at time $t_0 \geq t$ that will make $\overline{D}_1(t_0) = 0$. By assumption no job in $\hat{J}$ will make any new REQUEST after time t, so that by definition 1, $J_1$ will RELEASE all its resources at some finite time $t_1 > t_0$, thereby returning to the free state all resource elements controlled exclusively by $J_1$, plus any controlled in the shared mode by $J_1$ alone. This gives at time $t_1$:

$$\overline{RSHARE}(t_1) = \overline{RS}_2(t_1) = \overline{RS}_2(t)$$

$$\overline{RFREE}(t_1) = \overline{RFREE}(t) + \overline{AE}_1(t) + \overline{RSHARE}(t) - \overline{RSHARE}(t_1) = \overline{Z}_2(t) - \overline{RS}_2(t)$$

since no other $\overline{A}$ or $\overline{D}$ changed between t and $t_1$.

Now assume that jobs $J_1, J_2, \ldots, J_{i-1}$ have had their demands $\overline{D}_k(t)$ satisfied and have released their resources by time $t_{i-1}$, and that at

time $t_{i-1}$:

$$\overline{RSHARE}(t_{i-1}) = \overline{RS}_i(t)$$
$$\overline{RFREE}(t_{i-1}) = \overline{Z}_i(t) - \overline{RS}_i(t) \qquad \text{(III.C.5)}$$

We will show that this will also hold for $J_i$ at time $t_i \geq t_{i-1}$. Since assignments have been made only to jobs $J_k$ with $k < i$ by time $t_{i-1}$, $\overline{D}_p(t) = \overline{D}_p(t_{i-1})$, $\overline{A}_p(t) = \overline{A}_p(t_{i-1})$, and $\overline{RS}_p(t) = \overline{RS}_p(t_{i-1})$ for all $p \geq i$. Using the fact that jobs in F satisfy (III.C.2) at time t (by theorem 12), and the inductive conditions (III.C.5), we have:

$$\overline{DE}_i(t_{i-1}) = \overline{DE}_i(t) \leq \overline{Z}_i(t) - \overline{RS}_i(t) = \overline{RFREE}(t_{i-1}) + \overline{RS}_i(t) - \overline{RS}_i(t)$$
$$= \overline{RFREE}(t_{i-1})$$

$$\overline{DS}_i(t_{i-1}) + \overline{AS}_i(t_{i-1}) = \overline{DS}_i(t) + \overline{AS}_i(t) \leq \overline{Z}_i(t) - \overline{DE}_i(t)$$
$$= \overline{RFREE}(t_{i-1}) + \overline{RS}_i(t) - \overline{DE}_i(t)$$
$$= \overline{RFREE}(t_{i-1}) + \overline{RSHARE}(t_{i-1}) - \overline{DE}_i(t_{i-1})$$

which are just conditions (III.C.5) at time $t_{i-1}$.

Therefore, at time $t_{i-1}$, if $DNUMB_i(t_{i-1}) > 0$, an assignment can be made to $J_i$ that reduces $\overline{D}_i$ to zero. By assumption $J_i$ will not make any new REQUEST after time t, so that by definition 1, $J_i$ will RELEASE all its resources by some finite time $t_i \geq t_{i-1}$, thereby returning to the free state all resource elements exclusively controlled by $J_i$ plus any elements in the shared control mode that were controlled by $J_i$ alone. Therefore at time $t_i$, whether or not it was necessary to make an assignment to $J_i$ at time $t_{i-1}$, we will have for the value of $\overline{RSHARE}(t_i)$ the number of elements still under shared control by at least one job at time $t_i$, which will be

$\overline{RS}_{i+1}(t_i) = \overline{RS}_{i+1}(t)$, since for all $k \leq i$, $\overline{AS}_k(t_i) = 0$, whereas $\overline{AS}_k(t_i) = \overline{AS}_k(t)$ for $k > i$.

The value of $\overline{RFREE}$ at time $t_i$ is given by:

$$\overline{RFREE}(t_i) = \overline{RFREE}(t_{i-1}) + \overline{AE}_i(t_{i-1}) + \overline{RSHARE}(t_{i-1}) - \overline{RSHARE}(t_i)$$

$$= \overline{Z}_i(t) - \overline{RS}_i(t) + \overline{AE}_i(t) + \overline{RS}_i(t) - \overline{RS}_{i+1}(t)$$

$$= \overline{Z}_{i+1}(t) - \overline{RS}_{i+1}(t)$$

where we used relation (III.C.3) to define $\overline{Z}_{i+1}$ in terms of $\overline{Z}_i$.

Hence conditions (III.C.5) also hold for job $J_i$, the next job in F, at time $t_i \geq t_{i-1}$, which proves the induction. Q.E.D.

Theorem 15: If there exists a sequence F of all jobs in $\hat{J}$ that satisfies (III.C.2) at time t, then it is impossible to detect a deadlock for system $\hat{S}'$ at time t without further information on the resource requirements of jobs in $\hat{J}$. A system $\hat{S}'$ for which such a sequence exists is said to be deadlock-free at time t.

Proof:

Existence of such a sequence F means that deadlock is not inevitable at time t, since by theorem 14 there is an assumption about future job behavior which is sufficient for this F to be a finishing sequence for $\hat{S}'$. To disprove this assumption requires knowing at time t that some job in $\hat{J}$ will in fact make a request for additional resources at some time $t' > t$. Without this advance information on future resource requirements, it cannot be shown that F is not a finishing sequence for $\hat{S}'$ at time t, and hence it cannot be shown that $\hat{S}'$ is deadlocked at time t, by theorem 1. Q.E.D.

So that the early termination criterion used to derive $L_2$ from $L_1$ for deadlock detection in system $\hat{S}$ can be incorporated directly into $L_3$, the algorithm for deadlock detection in system $\hat{S}'$, we prove the following theorem which is analogous to theorem 11 in section II.K.

Theorem 16: Without advance information, a system $\hat{S}'$ that is deadlock-free at time t remains deadlock-free after an unsatisfiable REQUEST by job $J_k$ at time t if and only if there exists a partial sequence of jobs that satisfies (III.C.2) and ends with job $J_k$.

Proof:

Obviously if a partial sequence ending in $J_k$ does not exist, then a complete permutation sequence of all jobs in $\hat{J}$ that satisfies (III.C.2) also cannot exist, which implies that $\hat{S}'$ is deadlocked at time t, by theorem 13.

Suppose that at time t' immediately after the REQUEST is found to be unsatisfiable and $J_k$ is put asleep there does exist a partial sequence F' that satisfies (III.C.2) and ends in $J_k$. We show how to construct a complete permutation sequence of all jobs in $\hat{J}$ from this F'.

Let P be the set of jobs of $\hat{J}$ that are not in F', with all jobs renumbered so that $F' = \{J_1, J_2, \ldots, J_k\}$, and $P = \{J_{k+1}, J_{k+2}, \ldots, J_n\}$. Let F be a permutation sequence of all jobs in $\hat{J}$ satisfying (III.C.2) at time t when the REQUEST was invoked (by theorem 15 there must be at least one such sequence F if $\hat{S}'$ is deadlock-free at time t). Since the REQUEST was unsatisfiable at time t, the only change in data base $B_2$ between time t and time t' is that $D_{kj}(t') > 0$ for at least one j (and hence $DNUMB_k(t') > 0$), whereas $\bar{D}_k(t) = 0$ and $DNUMB_k(t) = 0$. $\overline{RFREE}$,

$\overline{RSHARE}$, all $\overline{A}_i$ and other $\overline{D}_i$ remain unchanged. Let job $J_i$ be the first job in F that also belongs to P. This defines a set Q of all jobs that precede $J_i$ in F, and a set of R of all other jobs in F (including $J_i$). Obviously any job in Q cannot be in P so that it must belong to F', making Q is a subset of F', and giving:

$$\sum_{J_p \text{ in } Q} \overline{AE}_p(t) = \sum_{J_p \text{ in } Q} \overline{AE}_p(t') \leq \sum_{J_p \text{ in } F'} \overline{AE}_p(t') = \sum_{p=1}^{k} \overline{AE}_p(t') = \sum_{p=1}^{k} \overline{AE}_p(t)$$

Using (III.C.3) to define $\overline{Z}$ and $\overline{RS}$, we get:

$$\overline{Z}_i(F,t) = \overline{RFREE}(t) + \sum_{J_p \text{ in } Q} \overline{AE}_p(t) + \overline{RSHARE}(t)$$

$$\leq \overline{RFREE}(t') + \sum_{p=1}^{k} \overline{AE}_p(t') + \overline{RSHARE}(t')$$

$$= \overline{Z}_{k+1}(F',t')$$

$\overline{RS}_{k+1}(F')$ is the number of resource elements shared by at least one job $J_p$ with $p > k$ (i.e., so that $J_p$ is in P), and $\overline{RS}_i(F)$ is the number of resource elements shared by at least one job $J_p$ such that either $J_p = J_i$ or $J_p$ follows $J_i$ in F (i.e., so that $J_p$ is in R).

Since $\overline{AS}_i(t) = \overline{AS}_i(t')$ for all i:

$$\overline{RS}_{k+1}(F',t') = \overline{RS}_{k+1}(F',t)$$

and

$$\overline{RS}_i(F,t) = \overline{RS}_i(F,t')$$

Since any job in P cannot be in Q, it must belong to R, making P a subset of R, which gives

$$\overline{RS}_{k+1}(F',t') = \overline{RS}_{k+1}(F',t) \leq \overline{RS}_i(F,t) \quad \text{or} \quad -\overline{RS}_i(F,t) \leq -\overline{RS}_{k+1}(F',t')$$

Since $J_i$ satisfied (III.C.2) as the $i^{th}$ job in F at time t, by theorem 12, we have:

$$\overline{DE}_i(t) = \overline{DE}_i(t) \leq \overline{Z}_i(F,t) - \overline{RS}_i(F,t)$$

$$\leq \overline{Z}_{k+1}(F',t') - \overline{RS}_{k+1}(F',t')$$

and

$$\overline{DS}_i(t') + \overline{AS}_i(t') = \overline{DS}_i(t) + \overline{AS}_i(t) \leq \overline{Z}_i(F,t) - \overline{DE}_i(t)$$

$$\leq \overline{Z}_{k+1}(F',t') - \overline{DE}_i(t')$$

Thus $J_i$ satisfies (III.C.2) as the $k+1^{st}$ job of a sequence constructed by appending $J_i$ at the end of F' to form a partial sequence F'' of k+1 jobs. To find the $k+2^{nd}$ job, remove $J_i$ from P to give P' and repeat the above proof for F' and P'. Continue until a complete sequence of all jobs in J is formed. This satisfies (III.C.2), so $\hat{S}'$ is deadlock-free at time t', by theorem 15. Q.E.D.

## D. Scheduler Primitives in $\hat{S}'$

The scheduler primitives in $\hat{S}'$ serve the same functions and are organized in the same hierarchical manner as were those in system $\hat{S}$. In fact, at the highest level, the job primitives, no changes in the definitions given in section II.H (figure II.H.1) are necessary, provided that in system $\hat{S}'$ the notation $\overline{N}$ represents a 2m-component vector whose first m elements are $\overline{NE}$ and whose second m components are $\overline{NS}$. Similarily set N will now contain two subsets, called NE and NS, which contain elements in either exclusive or shared control mode respectively. At the lowest level, only the primitives ALLOCATE and DEALLOCATE must be redefined to accept as an additional parameter a specification of the mode of control between the resource element and the job, which can be either exclusive or shared.

The major changes in the primitives must be made to ASSIGN and UNASSIGN, the intermediate level primitives, since this is where all the decision making and bookkeeping related to the true "scheduling" (including deadlock detection) occurs. The new primitive definitions are given in figure III.D.1, with the auxiliary functions PUTASLEEP and GETAWAKE in figure III.D.2. By comparing these definitions to those in figures II.H.2 and II.H.3 for system $\hat{S}$, it is apparent that the significant changes are: (1) the test in ASSIGN, which is based on conditions (III.C.1) rather than II.H.1; (2) the introduction of statements to handle the extra bookkeeping required by the new shared control mode possible in $\hat{S}'$; and (3) the introduction of statements in PUTASLEEP and GETAWAKE to maintain the data structures SC and $\overline{SCNUMB}$ needed by deadlock detection algorithm $L_3$. Theorem 17 proves that the new test in ASSIGN is the correct one, so that no deadlock is created if the test is satisfied and an assignment is made.

Theorem 18 proves that executing UNASSIGN whenever it is invoked will not cause a deadlock to arise.

Looking first at ASSIGN, it is clear that the bookkeeping for resources being assigned in exclusive control mode is identical to that for system $\hat{S}$ (figure II.H.2) in which this is the only possible mode of control. After these resources have been allocated, those in shared control mode must be allocated, and $AS_{kj}$ adjusted to reflect the fact. Statement S2 has been added to see if the total now assigned to $J_k$ in shared mode exceeds $RSHARE_j$, the number of different elements of type j that were previously owned in shared mode. If so, then elements that were previously free must be assigned to $J_k$ in shared mode to satisfy the demand $NS_j$. In this case $RFREE_j$ is decreased and $RSHARE_j$ increased by the number of elements needed from the free state in order to maintain the values defined by relation (III.B.2) and item 11. After this, job $J_k$ is allocated shared control of the $NS_j$ elements of type j, which completes the assignment.

The UNASSIGN primitive performs the complementary actions of ASSIGN, and once again the bookkeeping for releasing resources in exclusive control mode is identical to that for system $\hat{S}$ (figure II.H.2). After those resources are deallocated, the shared resources must be deallocated, and $AS_{kj}$ reduced by an appropriate amount. Statement U2 has been added to see if job $J_k$ is the last shared controller of a resource element $r_i$ of type j. If so, this element will return to the free state when deallocated by $J_k$, so that $RSHARE_j$ must be decreased by one and $RFREE_j$ increased by one to reflect this fact. In formulating the algorithm we have assumed that there exists a function "sharers" such that sharers ($r_i$) is the number of jobs currently sharing control of resource element $r_i$. After all the elements of type j in set NS have been checked and deallocated, the operation of UNASSIGN is finished.

Comparing the auxiliary functions in figure III. D. 2 with those in figure II. H. 3, the principal changes are: (1) the addition of the statement labelled P3 in procedure PUTASLEEP, and G3 in GETAWAKE, both of which are used to maintain the two new data structures SC and $\overline{\text{SCNUMB}}$; (2) the addition of the statement labelled P5 in PUTASLEEP and G5 in GETAWAKE, both of which maintain the QE and QS matrices.

In PUTASLEEP, if job $J_k$ does not possess shared control of any elements of type j, block P4 is not executed; otherwise, when $J_k$ is put asleep, the element of matrix SC corresponding to each of its shared resource elements must be incremented by one in order to preserve the correct value of SC as defined in item 12. Further, if the element of SC becomes non-zero in this process, then $\text{SCNUMB}_j$ must also be increased by one to indicate correctly the number of non-zero elements in column j of SC, as defined in item 13. This is performed in block P4, and is all that is required to maintain SC and $\overline{\text{SCNUMB}}$. The statement labelled P5 checks to see if a demand for elements of type j exists in either mode, and if so, block P6 is executed. In block P6, the job $J_k$ is enqueued in column j of QE and column j of QS, and since the ordering of elements in these two columns is independent (see definition 7, section III. E), this can be done by executing block P7 in parallel (column j+m of matrix Q is column j of matrix QS in our notation). Columns j and j+m of the D matrix are also updated in P7. A demand for elements of type j, regardless of the control mode desired, is entered in both the QE and QS matrices, but is counted only once in $\text{DNUMB}_k$. As before, the new deadlock detection algorithm, $L_3$, will be the definition of ENQUEUE($J_k$, $\overline{E}$).

Procedure GETAWAKE performs the complementary functions to PUTASLEEP. The statement G3 is added to maintain SC and $\overline{\text{SCNUMB}}$

correctly, and performs the reverse operations to P3 in PUTASLEEP. This is simply to decrease the value of $SC_{ij}$ for each element $r_i$ of type j controlled by $J_k$, since after the GETAWAKE, $DNUMB_k = 0$. Also, if this causes $SC_{ij}$ to go to zero, $SCNUMB_j$ must also be decreased by one as required by item 13.

Theorem 17: If conditions (III.C.1) are satisfied by some job $J_i$ with $DNUMB_i > 0$ at time t and $\hat{S}'$ is deadlock-free at time t, then $\hat{S}'$ will still be deadlock-free after an ASSIGN to $J_i$ at time t.

Proof:

Let F be a sequence of all jobs in $\hat{J}$ that satisfies (III.C.2) at time t (there must be at least one if $\hat{S}'$ is deadlock-free, by theorem 13), with the jobs numbered so that $J_k$ is the $k^{th}$ job in F. Since conditions (III.C.1) are satisfied by $J_i$, there are enough resources available to make an assignment to $J_i$, so that at time t' immediately thereafter we have:

$$\overline{RFREE}(t') = \overline{RFREE}(t) - \overline{DE_i}(t) - (\overline{RSHARE}(t') - \overline{RSHARE}(t))$$

$$\overline{D}_i(t') = 0$$

$$\overline{A}_i(t') = \overline{A}_i(t) + \overline{D}_i(t)$$

and

$$\overline{A}_k(t') = \overline{A}_k(t), \quad \overline{D}_k(t') = \overline{D}_k(t) \text{ for all } k \neq i.$$

Consider the sequence F' constructed from F by removing $J_i$ from its position in F and making it the first job in F', so that:

$$F' = \left\{ J_i, J_1, J_2, \ldots, J_{i-1}, J_{i+1}, J_{i+2}, \ldots, J_n \right\}.$$

Then F' satisfies (III.C.2) for system $\hat{S}'$ at time t', as is shown next.

Since $\overline{\text{RSHARE}}(t') = \overline{RS}_1(F', t')$ by (III.C.4), we have:

$$0 = \overline{DE}_i(t') \leq \overline{\text{RFREE}}(t') = \overline{\text{RFREE}}(t') + \overline{\text{RSHARE}}(t') - \overline{\text{RSHARE}}(t')$$

$$= \overline{Z}_1(F', t') - \overline{RS}_1(F', t')$$

Also $\overline{\text{RSHARE}}(t') \geq \max_k \overline{AS}_k(t') \geq \overline{AS}_i(t')$, by (III.B.1), so that:

$$\overline{AS}_i(t') = \overline{DS}_i(t') + \overline{AS}_i(t') \leq \overline{\text{RSHARE}}(t') \leq \overline{\text{RFREE}}(t') + \overline{\text{RSHARE}}(t')$$

$$= \overline{Z}_1(F', t') = \overline{Z}_1(F', t') - \overline{DE}_i(t')$$

Therefore $J_i$ satisfies condition (III.C.2) as the first job in F'.

Next consider any job $J_k$ with $k < i$.

$$\overline{Z}_k(F, t) = \overline{\text{RFREE}}(t) + \sum_{p=1}^{k-1} \overline{AE}_p(t) + \overline{\text{RSHARE}}(t)$$

$$= \overline{\text{RFREE}}(t') + \overline{DE}_i(t) + (\overline{\text{RSHARE}}(t') - \overline{\text{RSHARE}}(t))$$

$$+ \sum_{p=1}^{k-1} \overline{AE}_p(t) + \overline{\text{RSHARE}}(t)$$

$$= \overline{\text{RFREE}}(t') + \overline{AE}_i(t') - \overline{AE}_i(t) + \sum_{p=1}^{k-1} \overline{AE}_p(t') + \overline{\text{RSHARE}}(t')$$

$$\leq \overline{\text{RFREE}}(t') + \overline{AE}_i(t') + \sum_{p=1}^{k-1} \overline{AE}_p(t') + \overline{\text{RSHARE}}(t')$$

$$= \overline{Z}_{k+1}(F', t')$$

since $J_i$ is the first job in F'. At time t, $J_i$ followed $J_k$ in F, but at time t', $J_i$ precedes $J_k$ in F' and the position of all other jobs is unchanged. Hence the set of jobs following $J_k$ at position k+1 in F' is a proper subset of those following $J_k$ at position k in F, and this

implies:

$$\overline{RS}_k(F,t) \geq \overline{RS}_{k+1}(F',t') \qquad \text{or} \qquad -\overline{RS}_k(F,t) \leq -\overline{RS}_{k+1}(F',t')$$

Since F satisfies (III.C.2) at time t, we have:

$$\overline{DE}_k(t') = \overline{DE}_k(t) \leq \overline{Z}_k(F,t) - \overline{RS}_k(F,t) \leq \overline{Z}_{k+1}(F',t') - \overline{RS}_{k+1}(F',t')$$

and

$$\overline{DS}_k(t') + \overline{AS}_k(t') = \overline{DS}_k(t) + \overline{AS}_k(t) \leq \overline{Z}_k(F,t) - \overline{DE}_k(t)$$

$$\leq \overline{Z}_{k(1}(F',t') - \overline{DE}_k(t')$$

Thus $J_k$ satisfies (III.C.2) at time t' as the $k+1^{st}$ job in F', $k < i$.

Next consider any job $J_k$ with $k > i$.

$$\overline{Z}_k(F,t) = \overline{RFREE}(t) + \sum_{p=1}^{i-1} \overline{AE}_p(t) + \overline{AE}_i(t) + \sum_{p=i+1}^{k-1} \overline{AE}_p(t) + \overline{RSHARE}(t)$$

$$= \overline{RFREE}(t') + \overline{DE}_i(t) + (\overline{RSHARE}(t') - \overline{RSHARE}(t)) + \sum_{p=1}^{i-1} \overline{AE}_p(t')$$

$$+ \overline{AE}_i(t') - \overline{DE}_i(t) + \sum_{p=i+1}^{k-1} \overline{AE}_p(t') + \overline{RSHARE}(t)$$

$$= \overline{RFREE}(t') + \sum_{p=1}^{k-1} \overline{AE}_p(t') + \overline{RSHARE}(t')$$

$$= \overline{Z}_k(F,t') = \overline{Z}_k(F',t')$$

By construction, the same set of jobs precedes and follows $J_k$ in F' as precedes and follows $J_k$ in F, and since $\overline{AS}_p(t') = \overline{AS}_p(t)$ for $p \geq k > i$, we have $\overline{RS}_k(F,t) = \overline{RS}_k(F,t') = \overline{RS}_k(F',t')$ for job $J_k$ at position k in both F and F'.

Since $J_k$ satisfies (III.C.2) as the $k^{th}$ job in F, we have:

$$\overline{DE}_k(t') = \overline{DE}_k(t) \leq \overline{Z}_k(F,t) - \overline{RS}_k(F,t) = \overline{Z}_k(F',t') - \overline{RS}_k(F',t')$$

and

$$\overline{DS}_k(t') + \overline{AS}_k(t') = \overline{DS}_k(t) + \overline{AS}_k(t) \leq \overline{Z}_k(F,t) - \overline{DE}_k(t)$$

$$= \overline{Z}_k(F',t') - \overline{DE}_k(t') \quad .$$

Thus $J_k$ satisfies (III.C.2) at time t' as the $k^{th}$ job in F'.

Therefore F' satisfies (III.C.2) for all its jobs, which implies $\hat{S}'$ is deadlock-free by theorem 15. Q.E.D.

Theorem 18: If job $J_i$ issues a RELEASE at time t and $\hat{S}'$ is deadlock-free at time t, then $\hat{S}'$ will remain deadlock-free after an UNASSIGN from $J_i$ at time t.

Proof:

Let F be a sequence of all jobs in $\hat{J}$ satisfying (III.C.2) at time t (there must be at least one if $\hat{S}'$ is deadlock-free, by theorem 13), with the jobs numbered so that $J_k$ is the $k^{th}$ job in F. Consider the state of the system $\hat{S}'$ at time t' immediately after the UNASSIGN is complete:

$$\overline{RFREE}(t') = \overline{RFREE}(t) + \delta(NE) + (\overline{RSHARE}(t) - \overline{RSHARE}(t'))$$

$$\overline{D}_i(t') = \overline{D}_i(t) = 0$$

$$\overline{A}_i(t') = \overline{A}_i(t) - \delta(N)$$

and

$$\overline{A}_k(t') = \overline{A}_k(t), \qquad \overline{D}_k(t') = \overline{D}_k(t) \text{ for any } k \neq i \quad .$$

Then F will satisfy (III.C.2) for system $\hat{S}'$ at time t', as is shown next.

Consider any job $J_k$ with $k \leq i$.

$$\overline{Z}_k(F,t) = \overline{\overline{RFREE}}(t) + \sum_{p=1}^{k-1} \overline{AE}_p(t) + \overline{RSHARE}(t)$$

$$= \overline{\overline{RFREE}}(t') - \delta(NE) - (\overline{RSHARE}(t) - \overline{RSHARE}(t'))$$

$$+ \sum_{p=1}^{k-1} \overline{AE}_p(t') + \overline{RSHARE}(t)$$

$$\leq \overline{\overline{RFREE}}(t') + \sum_{p=1}^{k-1} \overline{AE}_p(t') + \overline{RSHARE}(t')$$

$$= \overline{Z}_k(F,t')$$

By definition, $\overline{RS}_k$ is the number of different resource elements controlled in shared mode by job $J_k$ or any job following $J_k$ in F, so that when $\overline{AS}_i$ decreases, $\overline{RS}_k$ may also decrease (it cannot increase) for all $k \leq i$. Therefore:

$$\overline{RS}_k(F,t) \geq \overline{RS}_k(F,t') \quad \text{or} \quad -\overline{RS}_k(F,t) \leq -\overline{RS}_k(F,t')$$

Since job $J_k$ in F satisfied (III.C.2) at time t, we have:

$$\overline{DE}_k(t') = \overline{DE}_k(t) \leq \overline{Z}_k(t) - \overline{RS}_k(t) \leq \overline{Z}_k(t') - \overline{RS}_k(t')$$

Since $\overline{AS}_k(t') = \overline{AS}_k(t)$ for $k \neq i$, and $\overline{AS}_i(t') = \overline{AS}_i(t) - \delta(NS) \leq \overline{AS}_i(t)$, we get:

$$\overline{DS}_k(t') + \overline{AS}_k(t') \leq \overline{DS}_k(t) + \overline{AS}_k(t) \leq \overline{Z}_k(t) - \overline{DE}_k(t) \leq \overline{Z}_k(t') - \overline{DE}_k(t')$$

Hence $J_k$ satisfies (III.C.2) at time t' as the $k^{th}$ job in F, $k \leq i$. Next consider any job $J_k$ with $k > i$.

$$\overline{Z}_k(F,t) = \overline{RFREE}(t) + \sum_{p=1}^{i-1} \overline{AE}_p(t) + \overline{AE}_i(t) + \sum_{p=i+1}^{k-1} \overline{AE}_p(t) + \overline{RSHARE}(t)$$

$$= \overline{RFREE}(t') - \delta(NE) - (\overline{RSHARE}(t) - \overline{RSHARE}(t')) + \sum_{p=1}^{i-1} \overline{AE}_p(t')$$

$$+ \overline{AE}_i(t') + \delta(NE) + \sum_{p=i+1}^{k-1} \overline{AE}_p(t') + \overline{RSHARE}(t)$$

$$= \overline{RFREE}(t') + \sum_{p=1}^{k-1} \overline{AE}_p(t') + \overline{RSHARE}(t')$$

$$= \overline{Z}_k(F,t')$$

Since $\overline{AS}_p(t) = \overline{AS}_p(t')$ for all $p \geq k > i$, $\overline{RS}_k(F,t') = \overline{RS}_k(F,t)$. Since $J_k$ satisfies (III.C.2) as the $k^{th}$ job in F at time t, we have:

$$\overline{DE}_k(t') = \overline{DE}_k(t) \leq \overline{Z}_k(t) - \overline{RS}_k(t) = \overline{Z}_k(t') - \overline{RS}_k(t')$$

and

$$\overline{DS}_k(t') + \overline{AS}_k(t') = \overline{DS}_k(t) + \overline{AS}_k(t) \leq \overline{Z}_k(t) - \overline{DE}_k(t) = \overline{Z}_k(t') - \overline{DE}_k(t') \quad .$$

Therefore all jobs in F satisfy (III.C.2) at time t', implying $\hat{S}'$ is deadlock-free at time t', by theorem 15.

Q.E.D.

```
ASSIGN(k, NE-bar, NS-bar)
    if NE-bar ≤ RFREE-bar and NS-bar + AS-bar_k ≤ RFREE-bar + RSHARE-bar - NE-bar
    then begin
        GETAWAKE(k, NE-bar, NS-bar);
        parallel for j in [1, m] do
S1:         begin
            AE_kj +:= NE_j;
            RFREE_j -:= NE_j;
            for p from 1 step 1 up to NE_j do ALLOCATE(J_k, j, exclusive);
            AS_kj +:= NS_j;
S2:         if AS_kj > RSHARE_j then
S3:             begin
                RFREE_j -:= AS_kj - RSHARE_j;
                RSHARE_j := AS_kj;
                end;
            for p from 1 step 1 up to NS_j do ALLOCATE(J_k, j, shared);
            end;
        end
    else PUTASLEEP(k, NE-bar, NS-bar);
```

Figure III.D.1 The Intermediate Level Primitives for System $\hat{S}'$

```
UNASSIGN(k, NE, NS)

    parallel for j in [1, m] do

U1:     begin

        AE_kj -:= δ(NE)_j;

        RFREE_j +:= δ(NE)_j;

        for each r_i of type j in NE do DEALLOCATE(J_k, r_i, exclusive);

        AS_kj -:= δ(NS)_j;

        for each r_i of type j in NS do

            begin

U2:         if sharers(r_i) = 1 then

U3:             begin

                RFREE_j +:= 1;

                RSHARE_j -:= 1;

                end;

            DEALLOCATE(J_k, r_i, shared);

            end;

        end;
```

Figure III. D. 1 continued

```
    procedure PUTASLEEP(k, NE-bar, NS-bar)
        if k not in E then
P1:         begin
            parallel for j in [1, m] do
P2:             begin
                RAVAIL_j -:= AE_kj;
P3:             for each r_i of type j controlled by J_k in shared mode do
P4:                 begin
                    if SC_ij = 0 then SCNUMB_j +:= 1;
                    SC_ij +:= 1;
                    end;
P5:             if NE_j > 0 or NS_j > 0 then
P6:                 begin
                    parallel for q := j, j + m do
P7:                     begin
                        D_kq +:= N_q;
                        QSIZE_q +:= 1;
                        ENQUEUE(J_k, Q-bar_q);
                        end;
                    DNUMB_k +:= 1;
                    end;
                end;
            COUNT +:= 1;
            ENQUEUE(J_k, E-bar);
            end;
```

Figure III.D.2 Auxiliary Scheduling Functions in System $\hat{S}'$

```
procedure GETAWAKE(k, NE, NS)
    if k in E then
G1:     begin
        parallel for j in [1, m] do
G2:     begin
        RAVAIL_j +:= AE_kj;
G3:     for each r_i of type j controlled by J_k in shared mode do
G4:     begin
        SC_ij -:= 1;
        if SC_ij = 0 then SCNUMB_j -:= 1;
        end;
G5:     if NE_j > 0 or NS_j > 0 then
G6:     begin
        parallel for q := j, j + m do
G7:     begin
        QSIZE_q -:= 1;
        DEQUEUE(J_k, Q_q);
        D_kq -:= N_q;
        end;
        DNUMB_k -:= 1;
        end;
    end;
    COUNT -:= 1;
    DEQUEUE(J_k, E);
    end;
```

Figure III. D. 2 continued

E. Algorithm $L_3$

Algorithm $L_3$, the algorithm for deadlock detection in system $\hat{S}'$, is given in figure III.E.1. The critical section procedure APPROVE used by $L_3$ is the same one used for algorithm $L_2$ and is not reproduced here (see figure II.K.1). The intuitive operation of $L_3$ is the same as for $L_2$, with the only changes being the extra bookkeeping and testing needed when there are two possible modes of resource control. This also creates the need for more working storage than was required for $L_2$. In addition to $\overline{\text{THRESH}}$, $\overline{\text{MARK}}$, $\overline{\text{OLDE}}$, and $\overline{\text{DN}}$ used by $L_2$, $L_3$ requires an s by m matrix TS, an m-component vector $\overline{\text{SCN}}$, and an n by m bit matrix X. TS is initialized with a copy of SC (see item 12), and is changed during the operation of $L_3$ as a means of keeping track of the number of controllers of a shared resource element. $\overline{\text{SCN}}$ is initialized to $\overline{\text{SCNUMB}}$, and will represent $\overline{\text{RS}}$ at each step of the algorithm. X is used to coordinate the actions of the algorithm on the two types (exclusive and shared) of demands on a single resource class.

Once again the ordering of the elements in the columns of Q is essential to the speed of the algorithm, making it proportional to n instead of $n^2$ if a search technique were used.

Definition 7: The ordering in system $\hat{S}'$ for elements in each column of Q is such that if k follows i as a value in column j of QE (QS) then:

$$DE_{ij} \leq DE_{kj} \quad (DS_{ij} + AS_{ij} \leq DS_{kj} + AS_{kj}) \qquad \text{(III.E.1)}$$

for all such i and k in column j. Further, if an index i appears as a value in column j of QE, it must also appear in column j of QS, and vice versa.

That this is the correct ordering to always guarantee that the "next" job is ready for testing by the algorithm can be seen from the next two lemmas

and the proof of theorems 19 and 20. As was true for the Q matrix in system $\hat{S}$, each column of QE (QS) is independent of all other columns in QE (QS), insofar as the indices which appear there and the relative order among these values is concerned. However, the corresponding columns of QE and QS are not independent, since if an index i appears in either, then it must appear in both, although the relative ordering between the items in column j of QE will not be the same as between the items in column j of QS, due to (III. E. 1). This interdependence of columns of QE with columns of QS was mentioned previously in section III. D, when the primitives to maintain QE and QS were defined, and is essential to the proof of the following lemmas and hence to the proper functioning of algorithm $L_3$.

Lemma 1: For fixed j, if index i is in column j of QE and job $J_i$ does not satisfy the conditions (for p = i):

$$DE_{pj} \leq Z_{qj} - RS_{qj}$$
$$DS_{pj} + AS_{pj} \leq Z_{qj} - DE_{pj} \qquad \text{(III. E. 2)}$$

then any job $J_k$ that follows $J_i$ in column j of QE and satisfies (III. E. 2) for p = k must precede $J_i$ in column j of QS.

Proof:

Assume $DE_{ij} > Z_{qj} - RS_{qj}$. Then from (III. E. 1) and the fact that $J_k$ follows $J_i$ in column j of QE, we have:

$$DE_{kj} \geq DE_{ij} > Z_{qj} - RS_{qj}$$

implying that $J_k$ could not satisfy (III. E. 2). But $J_k$ is assumed to satisfy (III. E. 2), so that we must have:

$$DE_{ij} \leq DE_{kj} \leq Z_{qj} - RS_{qj}$$

but since $J_k$ must satisfy (III. E. 2) and $J_i$ must fail, we also have:

$DS_{ij} + AS_{ij} > Z_{qj} - DE_{ij}$

and

$DS_{kj} + AS_{kj} \leq Z_{qj} - DE_{kj}$

Since $DE_{pj}$ is non-negative for all p and j, this gives:

$DS_{ij} + AS_{ij} + DE_{ij} > Z_{qj} \geq DS_{kj} + AS_{kj} + DE_{kj}$

Since $DE_{kj} \geq DE_{ij} \geq 0$, this gives:

$DS_{ij} + AS_{ij} > DS_{kj} + AS_{kj}$

which by definition 7 implies that $J_k$ must precede $J_i$ in column j of QS. (The same definition also insures that k and i will in fact be in column j of QS, since they are known to be in column j of QE.)

Q. E. D.

Lemma 2: For fixed j, if index i is in column j of QS and job $J_i$ does not satisfy conditions (III. E. 2) for p = i, then any job $J_k$ that follows $J_i$ in column j of QS and satisfies (III. E. 2) for p = k must precede $J_i$ in column j of QE.

Proof:

By definition 7, indices k and i are known to be in column j of QE, since they are both in column j of QS.

Suppose $J_k$ follows $J_i$ in column j of QE. Since $J_i$ does not satisfy (III. E. 2) but $J_k$ does, $J_k$ must precede $J_i$ in column j of QS, by lemma 1. Contradiction of the assumption of this lemma that $J_k$ follows $J_i$ in column j of QS. Hence $J_k$ must precede $J_i$ in column j of QE.

Q. E. D.

These lemmas show that with such an ordering in all columns of Q, if job $J_p$ in column j fails test (III.E.2) (which is just the $j^{th}$ component of condition (III.C.2)) and job $J_q$ in column j+m also fails (III.E.2), then no job $J_x$ following either $J_p$ in column j or $J_q$ in column j+m can pass the test. This is used in $L_3$ to obtain a linear scan down each column of Q, with no need for backup or repetitive searching.

Theorem 19: A permutation sequence F of all jobs in $\hat{J}$ that is composed of all jobs $J_i$ with $\overline{D}_i = 0$, followed by the list $\overline{E}$ generated by algorithm $L_3$, will satisfy (III.C.2).

Proof (by induction):

The inductive assumptions are that at the start of each execution of the statement labelled REPEAT in figure III.E.1: (1) the partial sequence F, consisting of the k jobs $J_i$ with $DNUMB_i = 0$ followed by the N jobs $J_i$ with $DNUMB_i > 0$ so far ordered in $\overline{E}$ by $L_3$, will satisfy (III.C.2); (2) the element in row i, column j of TS is the number of jobs not yet in F that have shared control of element i in resource class j; and (3) the following values are in $\overline{THRESH}$ and $\overline{SCN}$:

$$\overline{SCN} = \overline{RS}_{k+N+1}$$
$$\overline{THRESH} = \overline{RFREE} + \overline{RSHARE} + \sum_{i=1}^{k+N} \overline{AE}_i = \overline{Z}_{k+N+1} \qquad \text{(III.E.3)}$$

We have also assumed for notational convenience that at the start of each execution of REPEAT, the jobs in $\hat{J}$ are renumbered so that for $i \leq k+N$, $J_i$ is the $i^{th}$ job in F, and all jobs $J_i$ with $i > k+N$ are not yet in F. On the first execution of REPEAT, just after the initialization, only jobs $J_p$ with $p \leq k$ have $DNUMB_p = 0$. Using this and relations (III.B.3) and (III.C.3),

we have:

$$\overline{THRESH} = \overline{RAVAIL} = \overline{RFREE} + \overline{RSHARE} + \sum_{i=1}^{k} \overline{AE}_i = \overline{Z}_{k+1}$$

and (III. E. 4)

$$\overline{SCN} = \overline{RS}_{k+1}$$

since $\overline{RS}_p$ is defined as the number of different resource elements controlled by jobs following $J_{p-1}$ in a sequence, and this is the definition of SCNUMB (item 13) if the first p-1 jobs are the only jobs $J_i$ with $DNUMB_i = 0$. Finally, since TS is initialized to SC, and only jobs $J_p$ with $p > k$ have $DNUMB_p > 0$, it represents the number of jobs not in F with shared control of each resource element.

Therefore the inductive assumptions hold the first time if we define the partial sequence F at this point to consist of the jobs $J_p$ with $p \leq k$, and $N = 0$. These k jobs can be arranged in any order and still satisfy (III. C. 2), due to the fact that $\overline{D}_p = 0$ for $p \leq k$.

Now assume that (III. E. 3) holds for some $N > 0$. We will show that selection of the next ($N+1^{st}$) job by $L_3$ preserves the correctness of the inductive assumption when control returns to REPEAT.

Consider the block labelled L1 for some j. The test:

$$MARK_q \leq QSIZE_q \qquad \text{(III. E. 5)}$$

insures that block L2 is not executed if all $QSIZE_q$ elements in the $q^{th}$ column of Q have been tested and "approved" on previous iterations.

The test:

$$DE_{ij} \leq THRESH_j - SCN_j \quad \underline{\text{and}} \quad DS_{ij} + AS_{ij} \leq THRESH_j - DE_{ij} \qquad \text{(III. E. 6)}$$

is the $j^{th}$ component of condition (III. C. 2) for the $k+N+1^{st}$ job in F (due to the conditions (III. E. 3) for the value of $\overline{THRESH}$ and $\overline{SCN}$ at the start

of this iteration). Only if this test is true will the block labelled L3 be executed; if it is false, the execution of block L1 for this value of q terminates, with $MARK_q$ unchanged so that the next time L1 is executed with this value of j, the same job $J_i$ will be the first one tested. In block L3 the test:

$$X_{ij} > 0 \qquad \text{(III. E. 7)}$$

is necessary before the call to APPROVE because the demand for a resource of type j will be entered into column j of Q for those elements required in exclusive control mode, and into column (j + m) of Q for those in shared control mode. However both must be tested simultaneously (as in (III. E. 6)) and therefore should only be "approved" once for fixed i and j. $X_{ij}$ is initialized to 1 for all i and j, and after a call to APPROVE when the algorithm first encounters index i in either column j or (j + m), $X_{ij}$ is set to zero so that if and when i is encountered in the other column ((j+m) or j respectively), procedure APPROVE will not be called a second time for what is essentially the same demand.

Procedure APPROVE is identical to that used in algorithm L2. It decrements $DN_i$ by one, and if it goes to zero, then job $J_i$ is added to the list $\overline{E}$ and is therefore the next job in F (since $DN_i$ was initialized to $DNUMB_i$, when $DN_i$ is reduced to zero, all $DNUMB_i$ components of $(\overline{DE}_i + \overline{DS}_i)$ that are > 0 have satisfied (III. E. 6), and hence $J_i$ satisfies (III. C. 2) as the next job to be added to F). If i = k (where k is the job requesting the assignment), then F followed by $J_k$ is a partial sequence satisfying (III. C. 2) and ending in $J_k$, so that by theorem 16 system $\hat{S}'$ is deadlock-free. In this case the block A3 of APPROVE is executed to

construct the complete sequence in $\overline{E}$ from the previous sequence in $\overline{OLDE}$ (as described in the proof of theorem 16), and then the algorithm terminates. For each i and j, $DN_i$ is decremented at most once, since when control returns from APPROVE, $X_{ij}$ is set to zero and $MARK_q$ is incremented by one. Due to the resetting of $X_{ij}$, APPROVE will not be called for this value of i if it is encountered for either q = j + m or q = j. Since $\overline{MARK}$ is never decremented or reset in this algorithm, and since index i for job $J_i$ appears at most once in column 2 of Q, block L3 can never be entered more than once for this value of q and i. $MARK_q$ is incremented on every entry to block L3, whether or not it is necessary to call APPROVE, so that when control returns to the statement labelled AGAIN, it will point at the next element in column q of Q (or one beyond that if all elements in Q have been tested). Block L2 is thereby executed repeatedly for this value of q until either test (III. E. 5) or (III. E. 6) fails, at which time execution of block L1 for this value of q terminates. It is important to note that as soon as (III. E. 6) fails for some job $J_i$ in column j, and for some job $J_p$ in column j + m, there cannot be any job $J_x$ following $J_i$ in column j or following $J_p$ in column j + m that would not fail this test also, due to lemmas 1 and 2, and the ordering of elements in column j of both QE and QS.

When the statement labelled REPEAT has finished execution, the test at L4 is made. Since the algorithm did not terminate in procedure APPROVE on this iteration, job $J_k$ is not yet in F and hence the number N cannot have the value COUNT yet (as was possible in algorithm $L_1$ where the early termination criterion was not used). Therefore N is tested against OLDN, the value of N at the start of this iteration, to see if any job was added to

$\overline{E}$ on the iteration. If not, the algorithm terminates with a "deadlock" indication (it is shown in theorem 20 that $\hat{S}'$ is deadlocked when this happens). If N > OLDN then (N - OLDN) additional jobs were added to $\overline{E}$ on this iteration and block L5 is executed. This involves executing block L6 once for each of the jobs $J_i$ that were added to $\overline{E}$ on this iteration. The operations in this block accomplish two things: (1) all resource elements controlled by $J_i$ in exclusive control mode are added to $\overline{THRESH}$, so that

$$\overline{THRESH}_{new} = \overline{THRESH}_{old} + \overline{AE}_i = \overline{Z}_{k+OLDN+1} + \overline{AE}_i = \overline{Z}_{k+OLDN+2}$$

if $J_i$ is the k+OLDN+1$^{st}$ job in F; (2) for each element $r_p$ of type j controlled by $J_i$ in shared mode, $TS_{pj}$ is decremented by one, and if it goes to zero, then $SCN_j$ is also decremented by one. By assumption $TS_{pj}$ is the number of jobs not yet in F at the start of this iteration that had shared control of element p in class $R_j$. Therefore, if $J_i$ is one of these controllers, when it gets added to F, $TS_{pj}$ must be reduced to indicate one less controller not yet in F, and if it goes to zero, then all controllers must have been added to F, so that $SCN_j$ also is decremented by one to maintain in it the total number of elements of type j which are controlled in shared mode by at least one job not yet in F, which by definition is $RS_{k+N+2}$ after k+N+1 jobs are in F.

After block L6 has been executed for all values of q between OLDN+1 and N inclusive, the value in $\overline{SCN}$ is $\overline{RS}_{N+1}$, and the value in $\overline{THRESH}$

is:

$$\overline{\text{THRESH}} = \overline{\text{RFREE}} + \overline{\text{RSHARE}} + \sum_{i=1}^{k+\text{OLDN}} \overline{\text{AE}_i} + \sum_{i=\text{OLDN}+1}^{N} \overline{\text{AE}_i}$$

$$= \overline{\text{RFREE}} + \sum_{i=1}^{k+N} \overline{\text{AE}_i} + \overline{\text{RSHARE}}$$

$$= \overline{Z}_{k+N+1}$$

Since all the jobs so far in F satisfy (III.C.2), by construction, the inductive assumptions are true when control is returned to REPEAT at the start of the next iteration. Q.E.D.

Theorem 20: If algorithm $L_3$ fails to generate a permutation sequence F of all jobs in $\hat{J}$ satisfying (III.C.2), then no such sequence exists for $\hat{S}'$ at time t.

Proof:

Suppose $L_3$ fails to produce a sequence satisfying (III.C.2), but that such a sequence T does exist which satisfies (III.C.2) at time t. Let F be the partial sequence generated by $L_3$ up to the point it stopped, with the jobs renumbered so that $F = \{J_1, J_2, \ldots, J_k\}$, and define $P = \{J_{k+1}, J_{k+2}, \ldots, J_n\}$ as the set of the remaining jobs in $\hat{J}$. Obviously a job must be either in F or P but not both.

At the beginning of the iteration after the one that added $J_k$ to F, we have from conditions (III.E.3):

$$\overline{\text{THRESH}} = \overline{\text{RFREE}} + \overline{\text{RSHARE}} + \sum_{i=1}^{k} \overline{\text{AE}_i} = \overline{Z}_{k+1}$$

and $\overline{\text{SCN}} = \overline{\text{RS}}_{k+1}$

Failure of any $\text{DN}_i$ to be reduced to zero on the next iteration implies that for all jobs $J_i$ in P (i.e., not yet in F), at least one component j

of $\overline{D}_i$ fails to satisfy the test (III.E.6) (due to the ordering of columns of Q and lemmas 1 and 2), so that either:

$$DE_{ij} > THRESH_j - SCN_j = Z_{k+1,j} - RS_{k+1,j}$$

or (III.E.8)

$$DS_{ij} + AS_{ij} > THRESH_j - DE_{ij} = Z_{k+1,j} - DE_{ij}$$

or both. Let $J_i$ be the first job in sequence T that is also in P. This defines a set Q of all jobs that precede $J_i$ in T, and a set R of all other jobs in T (including $J_i$). Obviously Q is a subset of F, since no job ahead of $J_i$ in T is in P; and P is a subset of R. This gives:

$$\sum_{J_p \text{ in } Q} \overline{AE}_p \leq \sum_{J_p \text{ in } F} \overline{AE}_p = \sum_{p=1}^{k} \overline{AE}_p$$

If $J_i$ is the $q^{th}$ job in T, then $\overline{RS}_q(T) \geq \overline{RS}_{k+1}(F)$, or $-\overline{RS}_q(T) \leq -\overline{RS}_{k+1}(F)$, since the set R includes all jobs in P (i.e., not in F).

Since T satisfies (III.C.2) by assumption and $J_i$ is the $q^{th}$ job in T,

$$\begin{aligned}
\overline{DE}_i(t) &\leq \overline{Z}_q(T,t) - \overline{RS}_q(T,t) \\
&= \overline{RFREE}(t) + \sum_{J_p \text{ in } Q} \overline{AE}_p(t) - \overline{RSHARE}(t) - \overline{RS}_q(T,t) \\
&\leq \overline{RFREE}(t) + \sum_{p=1}^{k} \overline{AE}_p(t) + \overline{RSHARE}(t) - \overline{RS}_{k+1}(F,t) \\
&= \overline{Z}_{k+1}(F,t) - \overline{RS}_{k+1}(F,t)
\end{aligned}$$

and

$$\overline{DS}_i(t) + \overline{AS}_i(t) \leq \overline{Z}_q(T,t) - \overline{DE}_i(t) \leq \overline{Z}_{k+1}(F,t) - \overline{DE}_i(t)$$

But this contradicts condition (III.E.8) which must hold for every job in P, and $J_i$ is in P. Therefore, there cannot exist a sequence T satisfying (III.C.2) if algorithm $L_3$ fails. Q.E.D.

Theorem 21: Without advance information, algorithm $L_3$ determines uniquely whether or not system $\hat{S}'$ is deadlock-free at time t.

Proof:

If $L_3$ succeeds in generating a permutation sequence of all jobs in $\hat{J}$, this sequence will satisfy (III.C.2), by theorem 19, which implies that $\hat{S}'$ is deadlock-free at time t by theorem 16.

If $L_3$ fails to generate a sequence satisfying (III.C.2), then by the previous theorem no such sequence can exist for system $\hat{S}'$ at time t, which implies that $\hat{S}'$ is deadlocked, by theorem 13.

Q.E.D.

```
            begin
            parallel for j in [1, m] do
                begin
                THRESH_j := RAVAIL_j;
                SCN_j := SCNUMB_j;
                MARK_j := MARK_j+m := 1;
                parallel for i in [1, SCN_j] do TS_ij := SC_ij;
                parallel for i in [1, n] do X_ij := 1;
                end;
            parallel for i in [1, n] do
                begin
                DN_i := DNUMB_i;
                OLDE_i := E_i;
                Y_i := 1;
                end;
            N := OLDN := 0;
REPEAT:     parallel for j in [1, m] do for q := j, j+m do
L1:             begin
AGAIN:          if MARK_q ≤ QSIZE_q then
L2:                 begin
                    i := Q_MARK_q,q;
                    if DE_ij ≤ THRESH_j - SCN_j
                    and DS_ij + AS_ij ≤ THRESH_j - DE_ij then
```

Figure III. E. 1 Deadlock Detection Algorithm $L_3$

```
L3:             begin
                if X_ij > 0 then begin APPROVE(i); X_ij := 0; end;
                MARK_q +:= 1;
                go to AGAIN;
                end;
            end;
        end;
L4:     if N = OLDN then <deadlock> else
L5:         begin
            for q from OLDN+1 step 1 up to N do
L6:             begin
                i := E_q;
                parallel for j in [1, m] do
L7:                 begin
                    THRESH_j +:= AE_ij;
L8:                 if AS_ij > 0 then
                        for each r_k of type j controlled by J_i do
L9:                         begin
                            TS_kj -:= 1;
                            if TS_kj = 0 then SCN_j -:= 1;
                            end;
                    end;
                end;
            OLDN := N;
            go to REPEAT;
            end;
        end;
```

Figure III. E. 1 continued

F. Costs of Deadlock Detection in $\hat{S}'$

Since algorithm $L_3$ is similar to algorithms $L_1$ and $L_2$, and since the primitives in $\hat{S}'$ are similar to those in $\hat{S}$, the easiest way to analyze the cost of deadlock detection in $\hat{S}'$ is as a series of increases to the cost in system $\hat{S}$.

Analyzing $L_3$ in this manner we see first of all that, by comparing figure II.J.1 with (III.E.1), the initialization cost of $L_3$ is higher due to the need to initialize the m-component vector $\overline{SCN}$, the second m components of $\overline{MARK}$, the s by m matrix TS, and the n by m bit matrix X. These will all incur a constant cost, except for initializing TS, and since it is only performed once we will indicate it as $C_0'$, replacing the $C_0$ for system $\hat{S}$. Clearly $C_0'$ will be quite high, and is proportional to $m \times (s+n) + n$, whereas $C_0$ was proportional only to $m+n$. The cost $C_1'$ to execute block L1 in $L_3$ will be only marginally higher than $C_1$ for system $\hat{S}$, due to the replacement of test (II.J.7) with (III.E.6). However it must be multiplied by a factor of 2m rather than m, due to the doubling of the number of columns of Q. The same is true of $C_2$, although the factor of two really does not apply to the cost of executing APPROVE which is included in $C_2$. We will ignore this refinement, but cannot ignore the increased cost of the execution of block L5 caused by the introduction of the statement labelled L8 in figure III.E.1. Introducing a new cost $C_8$ as the cost of block L9 and separating this out from the cost of block L5 will leave $C_3$ relatively unchanged. The factor multiplying $C_8$ will then be $n \times m \times s$. Since APPROVE is unchanged in algorithm $L_3$, cost $C_7$ will be unchanged. We can also apply the averaging factors $\beta$, $\rho$, and $\mu$ exactly as for $L_2$, to give as a conservative estimate of the average cost of one execution of $L_3$: $C_0' + n \times \mu \times (C_7 + \rho \times C_3' + m \times \rho \times (s \times C_8 + 2 \times (C_1' + \beta \times C_2')))$ where

where $C_0' >> C_0$ but $C_1' \gtrsim C_1$, $C_2' \gtrsim C_2$, $C_3' \gtrsim C_3$. Clearly the cost of $L_3$ itself is proportional to n and m.

The cost of the overhead in the primitives required to maintain the data structures in $B_2$ used by $L_3$ will also be higher than that required to maintain $B_1$ for $L_1$ and $L_2$, as is expected. As before, all this extra bookkeeping is isolated in the procedures PUTASLEEP and GETAWAKE, and by comparing figure III.D.2 with figure II.H.3, we see that the major increase in cost will be due to the addition of statements labelled P3 and G3 in the new definitions. Letting $C_9$ be the cost of one execution of block P4 or G4 (they contain analogous operations), the maximum cost of executing P3 or G3 for fixed value of j is $s \times C_9$, which is extremely conservative since s is a maximum over all classes of resources. Hence the added total cost of both P3 and G3 together is $2 \times m \times s \times C_9$. Excluding this cost from the cost of executing block P2 or G2 will leave $C_5$ roughly unchanged, although it must now be multiplied by a factor of 2m rather than m for each procedure, since Q now has 2m columns instead of m, and block P7 (the major part of P2 after P3 is excluded) will be executed twice for each of the m values of j. The cost $C_4$, for one operation in ENQUEUE or DEQUEUE, is unchanged, but must be multiplied by an additional factor of two, since an index must appear in both matrix QE and matrix QS. Cost $C_6$ remains unchanged. Applying the average cost reduction factors $\alpha$, $\beta$, $\gamma$, and $\mu$, we obtain as the average overhead cost per REQUEST:

$$\alpha \times (2 \times m \times (s \times C_9 + 2 \times (n \times \beta \times \gamma \times C_4 + C_5')) + n \times \mu \times C_6)$$

where $C_5' \gtrsim C_5$.

A final cost reduction can be made by replacing s in the above expressions with z, the average number of resource elements in one class that are controlled in shared mode by a job. We then define the fraction $\sigma = z/s$ $(0 \leq \sigma \leq 1)$

as the average fraction of the elements in a resource class controlled in shared mode by an average job. This allows us to replace s by $\sigma \times s$, giving as the average cost per REQUEST for deadlock detection in $\hat{S}'$ with algorithm $L_3$:

$$\alpha \times (C'_0 + 2 \times m \times (2 \times C'_5 + s \times \sigma \times C_9) + n \times \mu \times (C_6 + C_7 + \rho \times C'_3$$

$$+ m \times (\rho \times s \times \sigma \times C_8 + 2 \times (\rho \times C'_1 + \rho \times \beta \times C'_2 + 2 \times \beta^2 \times C_4)) .$$

We must also consider the amount of additional storage needed to detect deadlock in system $\hat{S}'$, since it is considerably more than is necessary for a system $\hat{S}'$ that does not incorporate deadlock detection algorithms. It is also considerably more than was needed by the detection algorithms in system $\hat{S}$, due to the need for the two s by m matrices SC and TS. Clearly these need not be true rectangular matrices, since only the first $RMAX_j$ elements in the $j^{th}$ column are ever used. Therefore in an actual implementation, if the number of elements in different resource classes is vastly different, a packed representation would obviously be beneficial, even though the access mapping function would no longer be the simple matrix subscripting function.

The permanent data structures in $B_2$ solely for use in deadlock detection are the n by m matrices QE and QS, the s by m matrix SC, the n-component vectors $\overline{\text{RAVAIL}}$, $\overline{\text{SCNUMB}}$, $\overline{\text{QESIZE}}$, and $\overline{\text{QSSIZE}}$, and the n-component vector DNUMB, for a total of $2 \times n \times m + s \times m + 4 \times m + n$ storage cells. The working storage required by algorithm $L_3$ includes the m-component vectors $\overline{\text{THRESH}}$ and $\overline{\text{SCN}}$, the 2m-component vector $\overline{\text{MARK}}$, the s by m matrix TS, the n by m bit matrix X, and the n-component vectors $\overline{\text{DN}}$, $\overline{\text{OLDE}}$, and $\overline{\text{Y}}$, for a total working storage of: $4 \times m + s \times m + n \times m + 3 \times n$. Hence the total additional storage for deadlock detection in system $\hat{S}'$ with algorithm $L_3$ is: $4 \times n + 3 \times n \times m + (8 + 2 \times s) \times m$ cells.

## Chapter IV. Dynamic Deadlock Prevention

### A. Introduction

The model of resource allocation systems described in this chapter represents a second variation on the model presented in Chapter II. This modification consists of removing the condition "without advance information" from the results proven there, and demonstrating that, when some form of advance information is available, deadlock can be prevented rather than just simply detected. Following the lead of Habermann [11,12], we will require only the minimum amount of advance information that is sufficient to prevent deadlock. This will be in the form of a requirement on each job to specify in advance the maximum number of resource elements in each class that it will need to control at any single instant of time. We state this in the form of an assumption to be added to the list of five assumptions given in II. E.

Assumption 6: (Advance Information) At the time it is created, each job must specify a "maximum resource demand" for each resource class, such that at all times during its existence the job can request and be allocated at most the specified number of elements in each class.

This is the only advance information required of a job; no assumption is made about the sequence in which resources are required, or the duration of their use by the job.

The other five assumptions from section II. E are still valid, with a slight rewording of assumption 2 to conform with the requirements of assumption 6.

Assumption 2: (Non-virtual Resources) A job can specify as its maximum demand in each resource class at most all the resource

elements that exist in that class. There are no "virtual" resources accounted for in this model.

We use the notation $\hat{S}_a$ for the first type of system to be considered in this chapter, which is based on assumption 5 of section II. E, not assumption 5'. With this assumption resource elements can be controlled by only one job at a time; no simultaneous resource sharing is allowed. The second part of this chapter considers a system $\hat{S}'_a$ that is based on assumption 5', along with a version of assumption 6 modified to incorporate simultaneous resource sharing.

## B. Data Base $B_3$

The data base $B_3$ for system $\hat{S}_a$ is shown schematically in figure IV. B. 1. It is identical to data base $B_1$ for system $\hat{S}$, shown in figure II. F. 1, but with the addition of matrix C and vector $\overline{W}$, which are defined below.

The elements A, $\overline{RMAX}$, and $\overline{RFREE}$ in $B_3$ describe the current status of system $\hat{S}_a$ exactly as they did in $\hat{S}$, and remain as defined in Chapter II. $\overline{A}_i$ represents the number of elements of each type controlled by $J_i$ (all of which are assigned exclusively to $J_i$ since simultaneous resource sharing is not allowed in $\hat{S}_a$). $\overline{RMAX}$ represents the number of elements that exist in each resource class, and $\overline{RFREE}$ is the number of these currently in the free state. We must however distinguish two types of demands that exist in system $\hat{S}_a$: the current actual demand, and the current potential demand. In system $\hat{S}$ there was no advance information about resource requirements, so that there was no potential demand other than the current actual demand, and the D matrix represented them both simultaneously. In that system, if $\overline{D}_i = 0$, then job $J_i$ was able to progress without further scheduler intervention. There was no way for the scheduler to know whether or not job $J_i$ would require any additional resources at any time in the future.

In system $\hat{S}_a$ job $J_i$, at the time it is created, specifies a maximum demand vector $\overline{M}_i$ which will remain constant for the entire existence of the job. At any time $J_i$ can request and be allocated at most $M_{ij}$ resource elements of type j, and by assumption 2:

$$0 \leq \overline{M}_i \leq \overline{RMAX} \qquad \text{for all } i = 1, 2, \ldots, n. \qquad \text{(IV. B. 1)}$$

The D matrix is defined to be the potential demand matrix, such that:

$$\overline{D}_i(t) = \overline{M}_i - \overline{A}_i(t) \qquad \text{for all } i = 1, 2, \ldots, n. \qquad \text{(IV. B. 2)}$$

Thus $D_{ij}$ is the maximum number of resource elements of type j not yet allocated to $J_i$ at time t but which could be requested by $J_i$ at some future time in accordance with assumption 6. Clearly (IV. B. 1) and (IV. B. 2) together give:

$$0 \leq \overline{A}_i(t) + \overline{D}_i(t) = \overline{M}_i \leq \overline{RMAX} \qquad \text{for all } i = 1, 2, \ldots, n \qquad \text{(IV. B. 3)}$$

which is simply the requirement of assumption 6 in terms of the items in data base $B_3$. Since $\overline{M}_i$ is constant over the life of a job, and can always be computed from (IV. B. 3), there is no need to represent it in data base $B_3$, provided that at the time a job $J_i$ is created we require $\overline{D}_i = \overline{M}$ and $\overline{A}_i = 0$, which in effect says that at the time a job is created its potential demand is identical with its maximum possible demand.

To represent the current actual demand at time t we introduce the new matrix C.

Item 14: C--an n by m matrix, called the "current demand" matrix. If all the elements in the $i^{th}$ row of C are zero, then job $J_i$ is currently active. Otherwise job $J_i$ is in the waiting state due to an unsatisfied REQUEST for $C_{ij}$ resource elements of type j.

$\overline{C}_i(t)$ represents the unsatisfied resource demand of $J_i$ that does exist at time t (i. e. , that must be satisfied before $J_i$ can proceed), and $\overline{D}_i(t)$ is the demand which might exist at any time $t' \geq t$ (i. e. , that might have to be satisfied before $J_i$ can be guaranteed to return any resources, by assumption 4). This implies the following relationship:

$$0 \leq \overline{C}_i(t) \leq \overline{D}_i(t) \qquad \text{for all } i = 1, 2, \ldots, n. \qquad \text{(IV. B. 4)}$$

Using the redefined version of matrix D, all the previous definitions of the items $\overline{DNUMB}$, $\overline{E}$, COUNT, Q, and $\overline{QSIZE}$ in terms of D will remain identical, although the implications in terms of active and waiting states of a job are no

longer valid. That is, if $\overline{D}_i = 0$, then $\overline{C}_i = 0$, by (IV.B.4), and job $J_i$ must be active. But if $\overline{D}_i(t) > 0$, it is still possible for $\overline{C}_i(t) = 0$ to hold, in which case $J_i$ is active according to item 15. Since both $\overline{DNUMB}$ and $\overline{E}$ retain their original definitions in terms of D, it is no longer true that a job $J_i$ with $DNUMB_i > 0$ will necessarily be in the waiting state, and since the list of jobs $J_i$ in $\overline{E}$ will still be those with $DNUMB_i > 0$, list $\overline{E}$ no longer represents the list of waiting jobs, since some of the jobs on $\overline{E}$ may in fact be active. We therefore must introduce another n-component vector $\overline{W}$ to maintain a list of the indices of all jobs $J_i$ in the waiting state at time t (i.e., with $C_{ij}(t) > 0$ for at least one j). A job is put onto this list when it enters the wait state due to an unsatisfiable REQUEST, and is removed from $\overline{W}$ when it again enters the active state.

Item 15: $\overline{W}$--an n-component vector, called the "waiting list". It represents an ordered list of indices of jobs $J_i$ with $C_{ij}(t) > 0$ for at least one value of j.

Each of the jobs represented in the list $\overline{W}$ at time t will be in the wait state due to an unsatisfied current demand $C_i$. The ordering of the indices in $\overline{W}$ is irrelevant to the problem of deadlock prevention, provided the STARTUP procedure is as defined in figure IV.D.1. Depending on the ordering rule actually implemented, it may be preferable to represent $\overline{W}$ as a variable length list rather than a fixed length vector.

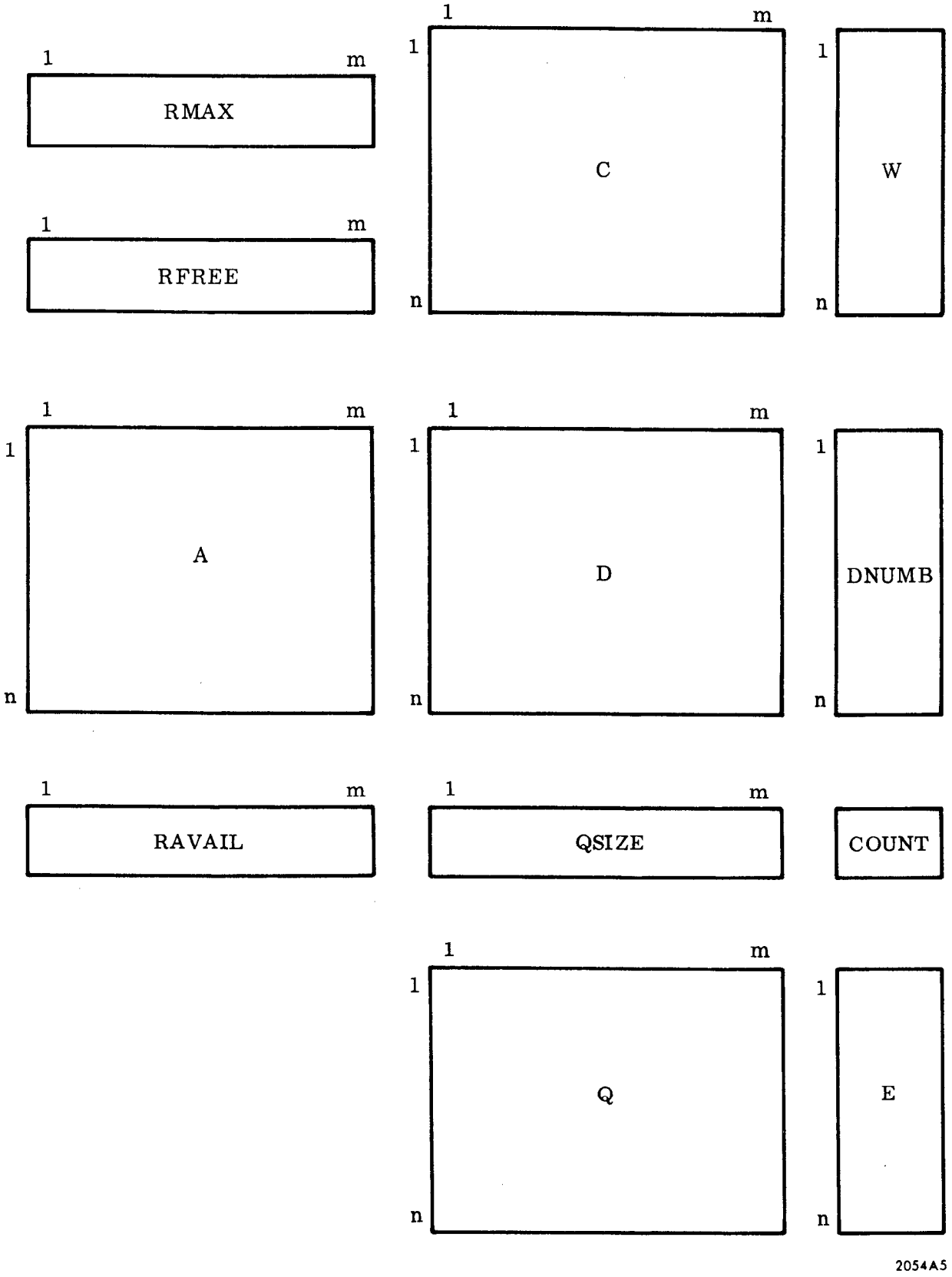


Figure IV.B.1 Scheduler Data Base $B_3$ for System $\hat{S}_a$.

## C. Formal Properties of $\hat{S}_a$

The new definition of matrix D does not invalidate definitions 1-5 of Chapter II, but simply requires that they be interpreted in a different manner for system $\hat{S}_a$ than for system $\hat{S}$. They now mean that a system $\hat{S}_a$ is considered to be deadlocked if the potential demand of at least one of the jobs in $\hat{J}$ cannot be satisfied in a finite time, and that a finishing sequence $\hat{S}_a$ is a permutation sequence of all jobs in $\hat{J}$ ordered according to the time at which their potential demands are met. The reasons for accepting such an interpretation are as follows:

We have demanded that, at the time it is created, a job must specify its maximum possible resource demand. This is the only advance information about job behavior the scheduler has, and is the minimum necessary to be able to prevent deadlock. Since the scheduler has no knowledge about the sequence in which resources are required by a job, and knows nothing about the length of time each resource will be controlled by a job, it has very little to go on in its task of preventing deadlock. Therefore, it must assume that the "worst possible case" will happen, which is that at some time t, all jobs will simultaneously demand all their resources, so that all the potential demands $\overline{D}_i$ become actual demands (i.e., $\overline{C}_i(t) = \overline{D}_i(t)$ for all i). If no deadlock is detectable with this assumption, then clearly no actual situation can be any worse in terms of resource demands. If deadlock is detected in the worst case, then since this was only a hypothetical situation, the scheduler can take steps to insure that it does not become a real situation. The whole philosophy of a deadlock prevention scheduler is to be able to detect deadlock in the worst case, so that all definitions and theorems about deadlock must be stated in terms of the worst case demand D.

With this revised interpretation of deadlock and finishing sequences it is clear that theorem 1 applies to system $\hat{S}_a$ as well as to $\hat{S}$, since it depends only on the formal definitions 1-5, and these remain unchanged (even if somewhat reinterpreted). With advance information in system $\hat{S}_a$ we are able to prove in the next theorem that conditions (II. G. 1) are not only necessary but are also sufficient for deadlock to be detectable in system $\hat{S}_a$. This is a stronger result than that of theorems 2 and 4, as is shown in the proof, and is exactly what is needed to be able to guarantee that a scheduler can foresee potential deadlocks and prevent them from developing into actual deadlocks.

It is important to note that whenever an assignment to job $J_i$ is made, $\overline{A}_i$ will be increased by an appropriate amount, $\overline{N}$, and due to relation (IV. B. 3), $\overline{D}_i$ will be decreased by exactly the same amount $\overline{N}$ (since $\overline{M}_i$ must remain constant in (IV. B. 3)). The fact that A and D always change by the same amount (in opposite directions) is the basis of the proofs of theorems 2 and 4, and since it remains true in system $\hat{S}_a$, these proofs will also remain valid.

Theorem 22: A permutation sequence F of all jobs in $\hat{J}$ is a finishing sequence for system $\hat{S}_a$ at time t if and only if the following conditions are satisfied by jobs in F:

$$\overline{D}_1(t) \leq \overline{RFREE}(t)$$
$$\overline{D}_i(t) \leq \overline{RFREE}(t) + \sum_{k=1}^{i-1} \overline{A}_k(t) \qquad \text{for } i = 2, 3, \ldots, n. \tag{IV. C. 1}$$

where we have assumed that the jobs are numbered so that $J_k$ is the $k^{th}$ job in F. Note that conditions (IV. C. 1) are identical to (II. G. 1).

Proof:

Only If: The "only if" part of this theorem is identical to theorem 2, and the proof remains identical since, for system $\hat{S}_a$, only if $\overline{D}_k = 0$ is it guaranteed that job $J_k$ will RELEASE all its resources in finite time, by definition 1.

If: The proof of the "if" part of this theorem is identical to that for theorem 4 for the following reason. The statement of theorem 4 said: "without advance information" we must make "the assumption that no new resource requests will be made at any future time". The "no new requests" in system $\hat{S}$ meant that once $\overline{D}_i$ is reduced to zero, job $J_i$ would not invoke the REQUEST primitive for more resources at any future time. With this hypothesis in the statement of that theorem it was shown in the proof that a permutation sequence satisfying (IV. C. 1) [(II. G. 1)] was a finishing sequence. However with the advance information required in system $\hat{S}_a$ it is known that once $\overline{D}_i$ is reduced to zero, $J_i$ cannot make any more REQUESTs for resources (since to do so would violate assumption 6). Therefore the condition hypothesized in theorem 4 for system $\hat{S}$ is always true for $\hat{S}_a$, due to the advance information, and the proof remains valid.

Q. E. D.

Since a system $\hat{S}_a$ is not deadlocked if and only if there exists at least one finishing sequence, by theorem 1, we can obviously combine this fact with the results of the previous theorem to get the following theorem, which is the counterpart for $\hat{S}_a$ to theorem 5 for $\hat{S}$.

Theorem 23: A system $\hat{S}_a$ is guaranteed not to be deadlocked at time t if and only if there exists a permutation sequence F of all jobs in $\hat{J}$

satisfying (IV. C. 1). A system $\hat{S}_a$ for which such a sequence exists at time t is said to be deadlock-free at time t.

## D. Deadlock Prevention in $\hat{S}_a$

### 1) Prevention via Detection

The objective of the scheduler S is to maintain system $\hat{S}_a$ in a deadlock-free condition at all times. With the additional advance information about maximum resource demands, it becomes possible to guarantee that an allocation will never be made that leads to a deadlock situation. For systems with no advance information, deadlocks could be detected only at the instant when they occurred, which was when a job made a REQUEST that could not be satisfied from the current free resources and had to be placed into the waiting state. If the system was previously deadlock-free, the entrance of a job into the waiting state might create a deadlock, and the detection algorithm had to be executed to decide. However, if the demand could be satisfied a new deadlock was impossible.

With advance information it is not the entry of a job into wait state due to insufficient free resources to satisfy a current demand, but rather the actual assignment of free resources to satisfy a current demand that can create a deadlock where none existed previously. Although this may seem strange at first, it becomes obvious when we note that since the definition of a finishing sequence is in terms of conditions that must be satisfied by D, any change in a value of D may cause a sequence F which previously satisfied (IV. C. 1) to no longer satisfy it. If a system $\hat{S}$ or $\hat{S}_a$ is known to be deadlock-free at some time t, a deadlock can be created where none existed before if a change is made to some $\overline{D}_i$ so that it becomes impossible to find a permutation sequence of jobs in $\hat{J}$ that satisfies (IV. C. 1).

In system $\hat{S}$, $\overline{D}_i$ changes only when a new REQUEST cannot be satisfied immediately (i.e., when a job is put asleep). In system $\hat{S}_a$, relation (IV. B. 2)

implies that the potential demand $\overline{D}_i$ changes only when $\overline{A}_i$ changes (i.e., when a job is allocated or deallocated control of resources). Deadlock is created in system $\hat{S}$ when a REQUEST is not satisfiable; it is created in $\hat{S}_a$ when a REQUEST is satisfied. (It is shown in theorem 26 that a RELEASE cannot create a deadlock in $\hat{S}_a$, and theorem 7 showed this was also the case in $\hat{S}$.) Another way to phrase it is that deadlock is created in system $\hat{S}$ when a job is put asleep; in $\hat{S}_a$ it is created when a job is gotten awake.

It is exactly the fact that deadlock is created in $\hat{S}_a$ only when a REQUEST is satisfied that enables the scheduler to prevent deadlock by simply not satisfying the REQUEST that creates it. The strategy for deadlock prevention in system $\hat{S}_a$ is as follows: if enough free resources exist to satisfy a current demand, change the state of the data base as if the demand were satisfied, then check to see if a deadlock is created. If no deadlock is detected, the job is allowed to proceed, since its demand is satisfied. If a deadlock is detected, simply restore the data base to its previous state (which is known to be deadlock-free) and then put the job asleep until some future time when more free resources become available.

In a sense our approach to deadlock prevention is to design a scheduler that detects deadlock when it occurs in a hypothetical worst case situation, and then is able to "recover" so that the worst case is never allowed to develop into an actual situation. Hence the deadlock is avoided. The recovery technique is simple and obvious, due to the fact that deadlocks can occur in $\hat{S}_a$ only when a current demand is satisfied: just postpone satisfaction of the demand until a more opportune time; let the job that made the REQUEST remain in the waiting state for however long it takes to guarantee that satisfaction of the REQUEST will not cause a deadlock. Note that the existence of at least one finishing

sequence for $\hat{S}_a$ at the time the REQUEST is made guarantees that the job will not have to wait forever, and that when job $J_i$ is put asleep $\overline{D}_i$ is not changed, so that the finishing sequence will also not change.

There are other recovery techniques to consider, such as preempting resources from other jobs (which is not possible in this model due to consequence 3). In this situation a scheduler should attempt to minimize the preemption costs (see [31]). Another approach would be to simply terminate the job that made the REQUEST causing the deadlock, although this would hardly be considered a "recovery" from the point of view of a system user.

The next theorem gives the formal proof that after a REQUEST by job $J_k$ is satisfied, the system $\hat{S}_a$ will remain deadlock-free if and only if there is a partial permutation sequence ending in $J_k$ and satisfying (IV. C. 1). This is the counterpart for $\hat{S}_a$ of theorem 11 for $\hat{S}$, and provides the formal basis for the assertion in the next section that algorithm $L_2$ as formulated for system $\hat{S}$ remains valid for system $\hat{S}_a$.

Theorem 24: A system $\hat{S}_a$ that is deadlock-free at time t remains deadlock-free after satisfaction of a REQUEST by job $J_k$ if and only if there exists a partial permutation sequence of jobs that satisfies (IV. C. 1) and ends with job $J_k$.

Proof:

If a partial sequence ending in $J_k$ does not exist, then a complete permutation sequence of all jobs in $\hat{J}$ that satisfies (IV. C. 1) cannot exist, which by theorem 23 means that $\hat{S}_a$ is deadlocked. At time t' immediately after the REQUEST (for $\overline{N}$ resource elements) is satisfied, suppose there exists a partial sequence F' that satisfies (IV. C. 1) and ends in $J_k$. We will show how to make this into a complete sequence of all jobs in $\hat{J}$, which by

theorem 23 makes $\hat{S}_a$ deadlock-free. Let P be the set of jobs of $\hat{J}$ that are not in F', with all jobs renumbered so that $F' = \{J_1, J_2, \ldots, J_k\}$, and $P = \{J_{k+1}, J_{k+2}, \ldots, J_n\}$. The complete sequence is constructed by repeatedly removing a job from P and adding it to the end of F' so that (IV.C.1) is satisfied at each step. At time t' we have:

$$\overline{A}_k(t') = \overline{A}_k(t) + \overline{N}$$

$$\overline{D}_k(t') = \overline{D}_k(t) - \overline{N}$$

$$\overline{RFREE}(t') = \overline{RFREE}(t) - \overline{N}$$

and $\overline{A}_i(t') = \overline{A}_i(t)$, $\overline{D}_i(t') = \overline{D}_i(t)$ for all $i \neq k$.

Let F be a finishing sequence for system $\hat{S}_a$ at time t (by theorem 1 there must be at least one such F if $\hat{S}_a$ is not deadlocked at time t), and let $J_i$ be the first job in F that also belongs to P. This defines a set Q of all jobs that precede $J_i$ in F. Obviously these jobs must also belong to F', by the definition of P and the selection of $J_i$. Therefore Q is a subset of F', and since $J_k$ is known to be in F', and may or may not be in Q, we have:

$$\sum_{J_p \text{ in } Q} \overline{A}_p(t) \leq \sum_{J_p \text{ in } Q} \overline{A}_p(t') \leq \sum_{J_p \text{ in } F} \overline{A}_p(t') = \sum_{p=1}^{k} \overline{A}_p(t) + N$$

Since $J_i$ satisfied (IV.C.1) as the $i^{th}$ job in F at time t,

$$\overline{D}_i(t') = \overline{D}_i(t) \leq \overline{RFREE}(t) + \sum_{J_p \text{ in } Q} \overline{A}_p(t)$$

$$\leq \overline{RFREE}(t') + \overline{N} + \sum_{p=1}^{k} \overline{A}_p(t') - \overline{N}$$

$$= RFREE(t') + \sum_{p=1}^{k} \overline{A}_p(t')$$

Hence $J_i$ satisfies (IV. C. 1) as the $k+1^{st}$ job to be added to F' to form a new partial sequence F'' of k+1 jobs satisfying (IV. C. 1). To find the next job to add to F'', simply remove $J_i$ from P to give P' and repeat the above proof for F'' and P'. Continue until a complete permutation sequence of all jobs in $\hat{J}$ is formed, and by construction this satisfies (IV. C. 1), so that $\hat{S}_a$ is deadlock-free, by theorem 23. Q. E. D.

2) Scheduler Primitives in $\hat{S}_a$

The scheduler strategy for deadlock prevention will require a major revision to the intermediate level primitives ASSIGN and UNASSIGN, since now it is no longer sufficient to merely detect creation of a deadlock; action must be taken to undo it. The new definitions of ASSIGN and UNASSIGN are given in figure IV. D. 2, with auxiliary functions in figures IV. D. 3 and IV. D. 4.

Figure IV. D. 1 describes the job primitives. These are nearly identical to the job primitives for system $\hat{S}$ in figure II. H. 1, with changes made only to the procedure STARTUP so that jobs in $\overline{W}$ rather than $\overline{E}$ are used as candidates to be awakened after an UNASSIGN, and so that the current demand C rather than D is used as a parameter to the ASSIGN. This is due to the obvious facts that $\overline{W}$ is the wait list in $\hat{S}_a$, and C is the current demand in $\hat{S}_a$, whereas $\overline{E}$ was both the wait list and finishing sequence and D both the current and potential demand in $\hat{S}$.

Looking first at the ASSIGN primitive, the test:

$$\overline{N} \leq \overline{RFREE} \qquad \text{(IV. D. 1)}$$

checks to see if enough free resources exist in each class to satisfy the current demand. If not, an assignment obviously cannot be made and PUTASLEEP is called to put the job to sleep on the list $\overline{W}$, if it is not already there. If (IV.D.1) is satisfied, then an assignment is possible and ASSUME is called to do the bookkeeping and allocate control of the resources to $J_k$. If the result of this is that $DNUMB_k = 0$ on return from ASSUME, then no deadlock is created (as is shown in theorem 25) and GETAWAKE is called to remove $J_k$ from the wait list $\overline{W}$. Otherwise, ENQUEUE($J_k$, $\overline{E}$), which as before will be the deadlock detection algorithm, is called to see whether the actions of ASSUME have created a deadlock. We suppose that ENQUEUE is designed to produce a "true" indication if no deadlock is detected and the sequence of jobs in $\overline{E}$, prefaced by all jobs $J_i$ with $DNUMB_i = 0$, satisfies (IV.C.1). In this case the assignment is completed by calling GETAWAKE to remove $J_k$ from the list $\overline{W}$, if necessary. If however the deadlock detection algorithm returns a "false", a deadlock is detected, and since none existed before, the allocation in ASSUME must have created it. Therefore UNASSUME is called to "undo" all the changes to data base $B_3$ performed in ASSUME, and then the job $J_k$ is put asleep on list $\overline{W}$ by calling PUTASLEEP. When a job $J_k$ is put asleep, only $\overline{C}_k$ changes; $\overline{D}_k$, $\overline{A}_k$, and $\overline{RFREE}$ are unchanged, so that the finishing sequence for $\hat{S}_a$ will not have to be changed either.

With this formulation of ASSIGN, the same deadlock detection algorithms $L_1$ and $L_2$ that were used to define ENQUEUE($J_k$, $\overline{E}$) in system $\hat{S}$ can be used in system $\hat{S}_a$ (modified to return "true" or "false" as described above). This is due to the fact that the conditions for deadlock to be detected in $\hat{S}_a$ are the same as in $\hat{S}$, as shown in theorem 23, and the partial sequence criterion for detection in $\hat{S}$ remains

valid in $\hat{S}_a$, as shown in theorem 24. Thus the bookkeeping operations performed in ASSUME and UNASSUME for $\hat{S}_a$ on Q, $\overline{QSIZE}$, $\overline{RAVAIL}$, D, and $\overline{DNUMB}$ are the same as those performed in GETAWAKE and PUTASLEEP in $\hat{S}$, although their order in the assignment sequence is different due to the changed interpretation of D.

The UNASSIGN primitive is considerably simpler than ASSIGN, since no decisions have to be made, as is shown in theorem 26. UNASSIGN calls procedure UNASSUME to perform the bookkeeping on data base $B_3$ necessary to indicate that resource elements are being released from $J_k$, and to deallocate each of these elements by calling DEALLOCATE to remove them from the control of $J_k$ and return them to the free state.

In procedures ASSUME and UNASSUME it is important to note the bookkeeping associated with D, Q, $\overline{DNUMB}$, and $\overline{RAVAIL}$, the items in data base $B_3$ used by the deadlock detection algorithms. In ASSUME, if $N_j > 0$ then, since $D_{kj} \geq N_j > 0$ must always hold or else assumption 6 is violated, job $J_i$ must be enqueued in column j of Q (by item 7). Therefore, when $D_{kj}$ changes value, $J_k$ must be removed from the position in column j of Q based on the old value of $D_{kj}$ and then, if the new value of $D_{kj}$ is not zero, reentered into the same column at a position based on the new value of $D_{kj}$. If $D_{kj}$ is zero, the job is not enqueued in the $j^{th}$ column of Q, and $DNUMB_k$ must be reduced by one since the number of non-zero elements in $\overline{D}_k$ is reduced by one (see item 5). After this has been done for all j elements of $\overline{N}$, $DNUMB_k$ is checked to see if it is zero, and if so, $\overline{A}_k$ must be added to $\overline{RAVAIL}$ in accordance with item 6.

Procedure UNASSUME performs the complementary actions to ASSUME. It also must reposition $J_k$ in column j of Q according to the new value of $\overline{D}_i$, and must maintain the correct value of $\overline{DNUMB}$ and $\overline{RAVAIL}$. It also calls

DEALLOCATE to remove the elements being released from the control of $J_k$, thereby returning them to the free state.

In system $\hat{S}_a$ all the overhead bookkeeping for the deadlock detection algorithms is embodied in the two new procedures ASSUME and UNASSUME, whereas the procedures GETAWAKE and PUTASLEEP have become extremely simple and are no longer concerned with the bookkeeping for deadlock detection, but merely maintain the current demand matrix C and the waiting list $\overline{W}$. This reflects the fact that in system $\hat{S}_a$ deadlock is created under different circumstances than it was in $\hat{S}$, and the fact that the scheduler in $\hat{S}_a$ uses the detection algorithm to prevent a deadlock from actually occurring.

3) Formal Properties

The next two theorems prove that an ASSIGN to $J_i$ that reduces $DNUMB_i$ to zero does not create a deadlock, and that an UNASSIGN can always be performed at the time it is invoked.

Theorem 25: If condition (IV. D. 1) is satisfied by some job $J_i$ with $DNUMB_i > 0$ at time t and $\hat{S}_a$ is deadlock-free at time t, then $\hat{S}_a$ will remain deadlock-free after an ASSIGN (ASSUME) to $J_i$ at time t, which reduces $DNUMB_i$ to zero.

Proof:

Condition (IV. D. 1) must be satisfied by a demand $\overline{N}$ of job $J_i$ in order to have enough resources of the proper type to make the assignment. At time t' immediately after the ASSUME of $\overline{N}$ resources we have $DNUMB_i(t') = 0$ which implies $\overline{D}_i(t') = 0$. Therefore we clearly have

$$\overline{D}_i(t') = 0 \leq \overline{RFREE}(t')$$

which means $J_i$ satisfies (IV. C. 1) as the first job in a partial sequence that